TUTORIAL

TUTORIAL GUIDE

Acknowledgements, Credits and Copyright

S-Base is published by:
Longman Logotron
124 Cambridge Science Park
Milton Road
Cambridge CB4 4ZS
Telephone (0223) 425558

Reference Documentation, S-Base Design & Programming: Simon Glass
Tutorial Documentation, Video & AddressBook Applications: Steve Simatos

S-Base
Copyright © Simon Glass, Steve Simatos & Logotron Ltd. 1992

ISBN 0582 21848 9

Please read carefully before using S-Base:

All rights are reserved. No part of this publication, either the whole or any part of the program or documentation, may be reproduced, transmitted, transcribed, stored in a retrieval system or translated into any language or computer language in any form or by any means electronic, mechanical, optical, chemical, manual or otherwise, without the prior express written permission of Logotron Ltd.

Logotron Ltd. provides the program S-Base under the rules of copyright, and licenses its use to the purchaser. This licence means that, unless you have Logotron's permission to the contrary:

a) you may use the program on a single computer, or move the program and use it on another computer, but you may not use the program on more than one computer at a time;

b) you may make one copy only of the program for backup purposes. Each disc is signed with a unique traceable number;

c) you may not make any copy of the documentation.

Disclaimer

No warranty is made with respect to the program, its quality or performance, its merchantability or fitness for any particular purpose. The program is sold 'as is'. You, the user assume responsibility for the selection of the program to achieve your intended results, and for installation, use and results obtained from the program.

Contents

1. Introduction

What is S-Base? 1
What can S-Base do? 1
Where do I start? 1
Conventions 2

2. Installation

Installation 3
Updating !System 3
Starting up 4

3. S-Base Applications

Resources 5
Data 6

4. Resources

Resource types 7
Data types 7
Files 9
Formats 11
Templates 13
Constants 14
Procedures 14
Programs 15
Handlers 16
Handler sets 16
Menus 17
Font styles 18
Sprites 19
Drawings 19
The next step 20

5. The S-Base environment

Starting up 21
Creating a new application 23
Managing applications 24
To follow 25

6. Editing Resources

Viewers 27
Window numbering 28

Opening and closing windows 28
Selecting objects 29
Renaming resources 29
Copying resources 30
Deleting resources 31
Dragging 33
Reordering resources 33
Copying between windows 34
Copying between applications 34
Outside S-Base 35
Printing viewers 36
Creating resources 36
Editing 37
Text editors 38
Font styles 38
Constants 40
Keyboard short cut summary 40

7. Variables and Expressions

What is a variable? 43
Using variables 44
Local variables 45
Arrays 46
Expressions and Operators 47
Operators 49
Functions 52
Managing variables 53

8. Files and File fields

Differences between fields and variables 55
Data types 56
Creating files 58
Using data types 62
The record size 62
The field order 62
The next step 63

9. Templates and Formats

Note for experienced users 65
What is a template? 66
Styles 66
Window styles 67

Icon styles 67
Adding a template 70
Editing templates 71
OS Units 75
The grid 75
Adding icons 76
Positioning icons 77
Altering icons 78
Style overrides 78
Resizing icons 79
Resizing templates 80
Creating formats 81
Automatic format generation 81
Element types 90

10. Cards

Controlling S-Base 95
Editing while S-Base is running 96
Comments 96
Creating cards 96
Creating handler sets & handlers 98
Attaching handler sets 100

11. Control structures

Control structure templates 103
Case..when..endcase 104
If..then..endif 106
For..next 108
Repeat..until 109
While..endwhile 110
Def proc..enddef 111
Which control structure? 112

12. Indexes

Adding indexes 115
Indexing on more than one field 116
Traversing an index 117
Index and file numbering 118
Finding records 118

13. More on cards

Scared cards 121

Nested cards 122
Keypress handlers 124
Confirmed exit 125
Protected icons 127
Protected fields 130

14. Program menus

Creating a program menu 133

15. Reports

Creating reports 135
Report layout 137
Element ordering 141
Request entry handlers 142
Running reports 143
Controlling reports 144
Text-only reports 145
Formatting expressions 145

16. Dates and times

Date and time functions 149
Date and time arithmetic 150

17. Mathematics functions

Trigonometry functions 153
Statistical functions 154
Number conversion functions 154
General mathematics functions 155

18. Strings

The basics 157
String functions 158
@str 159
Strings in formats 160

19. Advanced reports

Multi-column reports 163
Multi-entry reports 166

20. Handlers and handler sets

S-Base handlers 169
Handler functions 171
The default event handler 172
Raw event handlers 173

21. Advanced Cards

Selectors 175
Selecting multiple items 177
Concurrent polling 179

22. Menus

The menu resource 183
Creating menus 183
Menu maker handlers 185
Menu select handler 187
Altering menus 188
Icon bar menu 188

23. Tables

Creating tables 191
Request entry handler 194
Exit entry handler 195
Using tables 195
Attaching tables to cards 196
Protected Fields 197
Table extension 197
Table extension handler 198
Table insertion and deletion 200

24. Advanced files

Floating structures 203
Variable length records 207
Zero length structures 209
Select 210
Filters 211
File commands and functions 212
Progress functions 213
Importing and Exporting 214

25. External data

Importing external data 219

26. Graphics

Graphics commands 221
Redraw handlers 221

27. Accessing RISC OS

Sys 225

Error handling 226

28. Archiving and Importing applications

Archiving 229

Importing 230

Index **231**

1. Introduction

What is S-Base?

S-Base is a database system used to develop applications that process information.

S-Base contains its own language, called *S*, which allows you to give instructions to S-Base. This programmability makes S-Base very flexible.

The *S-Base Developers Environment* (SDE) allows you to create and edit applications. You can also run them to try them out.

What can S-Base do?

S-Base can make the job of writing sophisticated database applications easy. The SDE contains a powerful array of tools designed to turn your ideas into reality.

The secret of S-Base's power lies in its extensive language which reduces the complex operations of RISC OS and database management into simple commands. The advanced user interface allows you to produce professional results in no time.

Where do I start?

This tutorial aims to give you a good working knowledge of S-Base with the minimum of pain. It is designed as a 'work through' tutorial. Each chapter relies on exercises completed in previous chapters. You should read this tutorial in sequence.

Due to the large number of facilities in S-Base you will find that you will have to read and follow a fair amount of this tutorial before you get the satisfaction of completing your own application. But this small initial investment will pay dividends later: you will have fewer misconceptions and greater understanding of how S-Base works.

This tutorial discusses the basic facilities of S-Base and will describe the construction of a simple address book. Many other features of S-Base are illustrated in Video, the demonstration application supplied with S-Base.

Because of the complexity of S-Base and its language, it may be necessary to consult the S Reference from time to time for further information on commands and functions. This tutorial will only cover material used in the

application you will be constructing.

Conventions

You should refer to your computer's Welcome Guide and be sure you understand the basic concepts of RISC OS before attempting to follow this tutorial.

The following conventions are used in this guide:

Select refers to the left button on the mouse.

Menu refers to the middle button on the mouse.

Adjust refers to the right button on the mouse.

Click refers to a single click with one of the mouse buttons (e.g. Click Adjust).

Double-click refers to two clicks in quick succession.

Shift-double-click refers to two clicks in quick succession while the Shift key is depressed.

Ctrl-double-click refers to two clicks in quick succession while the Ctrl key is depressed.

Text which should be entered from the keyboard or is part of an application is shown `in this font.`

`s:` refers to text which should be entered at the S-Base *command line*.

Bold text signifies an S command or function.

Italic text signifies a new concept.

Since there is often a large amount of text between modifications to the tutorial application, this shading is used to signify changes to the tutorial application. You should not skip through reading only the grey sections, but it may be useful to be able to skip back and find a change you made earlier.

2. Installation

Before opening the disc envelope, you are reminded of the licence conditions attached to using S-Base. The Single User Licence Agreement allows S-Base to be used on a single computer at a time. A backup copy of the software may only be made for the purposes of protecting yourself against accidental damage to the original discs as supplied in the package. The copyright holders WILL take legal action against anyone found in possession of unlicenced software or found to be in breach of the licence agreement.

You are also reminded to complete the registration card included with the package. Longman Logotron will only provided support to registered users or users who can provide proof of purchase. Support includes provision of any upgrades to the software that may appear in the future.

Installation

A hard disc is recommended for serious use of S-Base although it will run correctly from a floppy drive. Hard disc users should create a separate directory for S-Base and copy the contents of the supplied discs into this directory.

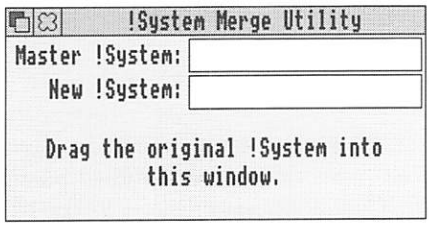
Floppy disc users should copy S-Base onto one or more 'working discs'.

In either case, you should keep your original S-Base discs safe in case your working copy is damaged or lost.

Updating !System

You will already have a copy of !System which is required by many RISC OS applications. S-Base is supplied with an update for your !System should you require it.

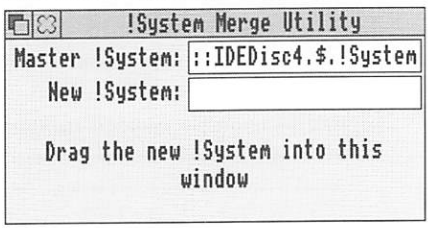
!SysMerge is supplied on the S-Base discs to help you upgrade the !System on your master disc. Starting this application will present the SysMerge dialogue:



Now follow these steps to update your !System:

1. Locate your existing !System (the 'Master !System'). If your Master !System is stored on a floppy disc, you will have to insert this disc into the drive.

2. Drag this !System into the SysMerge dialogue.



3. Locate the !System supplied with S-Base (the 'New !System'). You will have to insert the S-Base System disc into a floppy drive.

4. Drag this !System into the SysMerge dialogue.

5. SysMerge will then start work, updating any modules which are older than the modules supplied with S-Base.

6. Click Menu in the SysMerge dialogue and select Quit.

Note that if you have more than one disc containing !System, all discs must be updated as described above.

Starting up

To start S-Base, double-click on the !S-Base application. S-Base will then install itself on the icon bar. To quit, select Quit from the icon bar menu.

You are now ready to start.

3. S-Base Applications

The S-Base Developers Environment (SDE) allows you to edit S-Base *applications*. These applications look like normal applications, with an icon and an exclamation mark at the start of the filename, though you will need S-Base to run them. Double-clicking on an S-Base application will load it into the SDE ready for you to begin work.

An S-Base application consists of two components. *Resources* are set up using the SDE. They tell S-Base how to manage and display your application. Once your application is finished the resources do not alter.

Data is entered by users of your application and can be loaded, processed, and stored using the resources you have defined. This is not fixed and can be altered at any time. Only the structure of the data is fixed by the resources. S-Base sets up default data when required by your application.

Resources

RISC OS applications (like S-Base itself) are generally developed using a language called C. With C you write a large complicated program with perhaps the help of a screen designer. S-Base lets you develop applications as a set of related resources. There is no large chunk of program code to delve into, just a number of different types of resources.

Although there are thirteen different resource types they usually take up much less disc space than the data. The function of resources varies greatly; in general they are used to manage data, control the screen and printer, and handle input from the mouse and keyboard. These will be described in more detail in the next chapter.

All resources have a *label* which allows S-Base and yourself to identify one from another. Each label may contain up to 31 characters and must be unique to that resource type.

The SDE allows fully integrated editing of all resources through its own editors. These link each resource together, with internal checking to reduce the risk of errors creeping in. Should you delete or alter a resource, the SDE will update all references to that object throughout your entire application.

The SDE also allows you to load more than one application at a time into memory and copy resources between them.

Application resources can be archived as a single text file and sent via modem or disc to another user, or imported back into S-Base. The application archive can be edited with a simple text editor.

The SDE can automatically create some resources from others. For example, after defining the structure of a file you can ask S-Base to create a format for display of that information. Of course, you can then change that format.

Data

An average S-Base application exists simply to process data entered by the user. The format of your data is defined in the resources using the SDE, though the data itself is stored separately in a directory inside your application.

To speed up searching through the data, S-Base maintains various *indexes* into your data just like an index in a book. These allow you to quickly look up data or print out a list in a particular order. For example, S-Base can answer the question, 'Does Joe Bloggs have any overdue videos?', quickly rather than having to look through each video cassette to see if it has been issued to Joe Bloggs and is overdue.

Like data files, indexes are stored in the data directory of your application and are defined by you in the resources.

The next chapter will give you a brief description of the resources in S-Base. A more detailed look at these resources is given in later chapters of this tutorial.

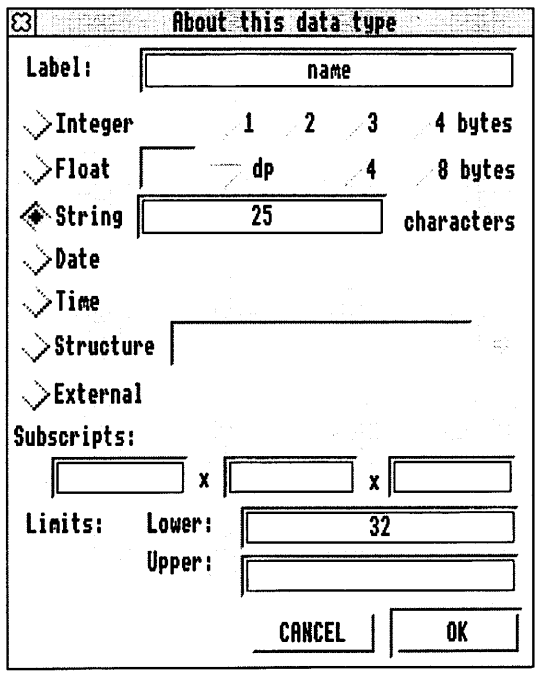
4. Resources

This chapter will help you understand exactly what makes up an S-Base application. Once you have read this you will be ready to start up S-Base and try some things out.

Resource types

Resources define the structure and operation of an S-Base application and are created using the SDE. The thirteen different resources control various aspects of your application from screen presentation to advanced mathematical calculations. This chapter gives an overview of the S-Base resource types and how each is used. A sound knowledge of the contents of this chapter is assumed in the remainder of this tutorial.

Data types



A *data type* describes the composition of a piece of data just as a chemical equation describes a molecule. Think of a data type as a description of the place where the data is stored. There are seven basic data types available:

Integer – a value with no fractional part. This can store numbers like 5 and –12, but not 45.67. An integer can be stored in 1, 2, 3 or 4 bytes. The largest size can hold numbers from about –2000 million to 2000 million.

Float – a value with a fractional part. This can be either single (4 bytes) or double (8 bytes) precision. A float is an approximation to a real value. For example, a float can store the value 5.9 or –126.554. The precision refers to the accuracy with which the value is stored. Single precision (4 bytes) allows about 6 significant digits to be stored accurately (eg 123.056, 0.000123056). Double precision allows 14.

String – a character value of predefined length. For example, a string of length 6 could store 'S-Base'. Strings have no numerical value and are used to store names and addresses or ant other text.

Date – days, months, and years. Accurate to one day and capable of storing all reasonable AD dates. You use a date for recording when an invoice was raised, for example.

Time – stored in seconds. Accurate to one second and capable of storing dates from 1970 to around 2100. Times are used when more accuracy is required than a date provides.

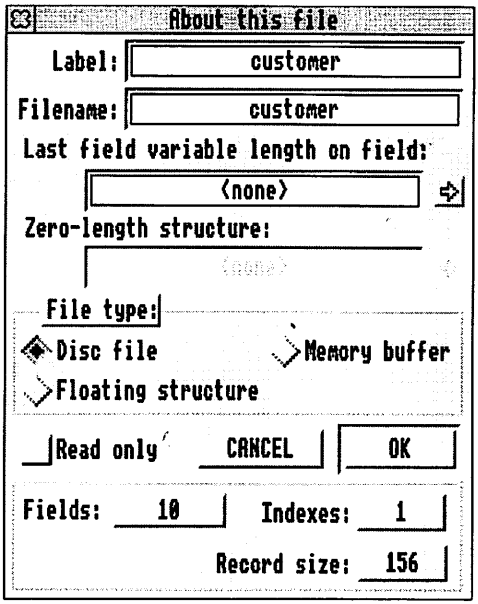
Structure – groups of fields or a file. A structure can hold a group of different data types. For example, an address structure might hold four strings, representing four lines of an address.

External – a value not stored in the database itself, but in an external file. An external value can be of any size and type. The external data type is often used to store drawings and sprites.

Data types are used as the building blocks for designing data files.

When running your application, S-Base will sometimes create data types for use by the system. These 'system data types' have labels surrounded by curly braces {}. They appear in the data type list but are not selectable from menus and can be safely ignored. If you delete a system data type, S-Base will automatically recreate it as required.

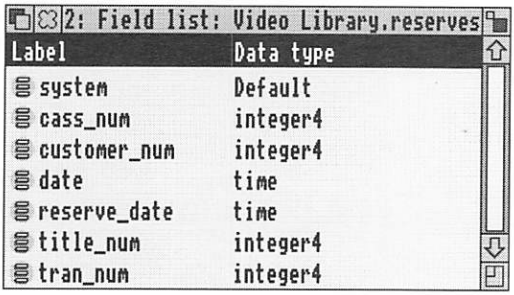
Files



The files resource is the blueprint used by S-Base to decide how data is to be stored on disc or in memory. It also gives the order in which you would like your data to be retrieved for printing or editing. This resource is what makes your application a database application.

The structure of a file is defined using *fields*. A field is a unit of data which you have designed using a data type. You can group up to 254 fields of any type into *records* for storing in your file. While your application is running, the user can create any number of records for that file with each having the same structure. The *record size* for the entire file is the sum of all the field sizes for that file.

A typical set of file fields is shown below. You can see that this file (called `reserves`) contains seven fields, with the data types listed in the second column. The first field in the file has the label `system` and is created automatically by S-Base when you add a file. This field is used by S-Base to determine whether a record has been deleted and should not concern your application.



Files are stored inside your application's data directory with a separate directory for each. A limit of 10 characters per disc file label is imposed under ADFS but other filing systems may impose different restrictions.

S-Base supports up to 254 files in an application of which any or all can be active at one time. However, in practice the number of disc files may be limited by your computer's filing system, or available memory.

There are no set limits to the number of records in a file or the amount of data in each record except that S-Base must be able to fit one record from each file in memory. Thus you are restricted to the memory available in your computer.

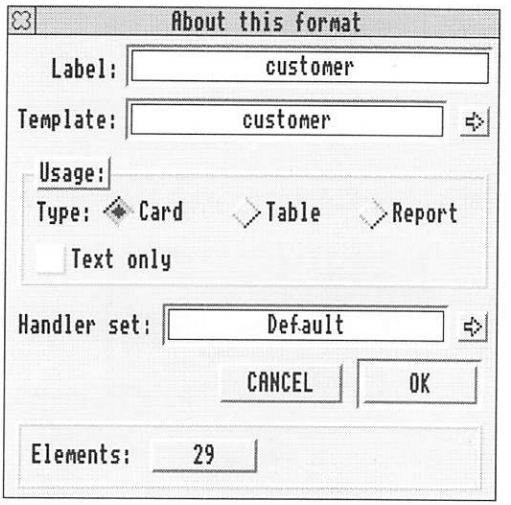
There are several different types of file that you can define in the SDE. The most commonly used files have *fixed length records* of identical length. Other files contain *variable length records* with different amounts of data in each record. Some files exist only in memory and are not stored on disc at all. Files can be *read only* (for example, a file on a CD–ROM).

Within applications, a file field is referenced by the file name followed by a dot and your field name. For example `video.title` refers to the `title` field of the `video` file.

Indexes are an important part of files. These are used to define the order in which data is retrieved from a file and significantly increase the speed of searching on the database. For example, you might like to index video titles in alphabetical order. A file may have any number of indexes.

A detailed look at all aspects of files and indexes appears later in this tutorial.

Formats



A format defines the layout for entering and displaying data on the screen or printer. Formats are made up of *elements* similar to the way that files are made up of fields. There are nine format element types:

Henry

A *field* is a place where the user can enter data into the application. It can contain data of any type with its behaviour varying with each type.

Cancel

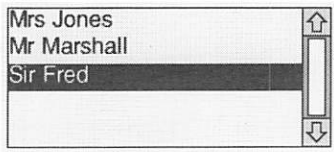
An *icon* is a button which the user can click on to control the application. This is directly linked to an icon in a template (described later).

Given name:

A *prompt* is some text used for display only, to prompt the user. It cannot accept the cursor.

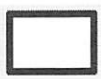
A *sprite* is a small picture used mainly to enhance the appearance of the screen. Sprites are created with applications such as Revelation and Paint. Sprites are stored and selected from the sprite resource.

A *drawing* is a detailed picture scalable to any size. It is mainly used to enhance the appearance of the screen. Drawings are created with applications such as !Draw. Drawings are stored and selected from the drawing resource.



A *selector* is a scrollable menu embedded in the format, from which the user can select one or more items.

An *area* (the darker box underneath the three fields) is a part of the format designated for a special purpose. For example the area above is an entry in a report.



A *rectangle* is a coloured box used to enhance the appearance of the screen. It can be filled and use a number of corner styles.

A *line* is a coloured line used to enhance the appearance of the screen. It can vary in thickness and style.

There is no limit to the number of elements in a format except good design principles. Massive formats with huge numbers of elements will confuse and irritate the user no matter how many pretty lines and rectangles you add.

There are three types of format: cards, tables and reports. *Cards* generally contain elements associated with a single record from a file and may be

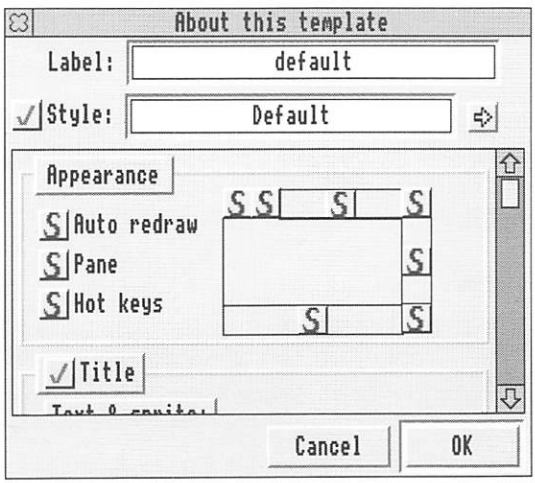
positioned anywhere on the page.

Tables allow the user to display or enter data from multiple records in one window. One record is displayed per entry with a tabular layout similar to that of a spreadsheet.

Reports may look similar to both cards and tables but are used only for output to screen or printer. These can be *text only*, which means they will appear as text output in the printer's native font. This is useful for printing large lists on a dot matrix printer. Dot matrix printers tend to be slow when outputting RISC OS fonts as graphic dumps.

S-Base manages *events* associated with formats, such as keypresses and mouse clicks, through *handler sets*. These are attached to a particular format and are called by S-Base so that your application can control or process data that is being entered by the user. If no handler set is attached to a format, S-Base handles all events automatically.

Templates



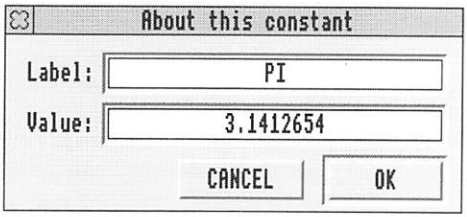
A *template* defines the size and look of a window. Most templates contain icons for the user to control an application, while some may be blank and used solely to define the size and shape of a window. Templates are used by almost all RISC OS applications and help developers to design window layouts quickly and easily.

S-Base uses templates to define the size, shape and title of your formats.

Icons defined in templates directly relate to the icon elements in formats, and are usually placed on top of each other, for reasons explained later.

S-Base provides an easy-to-use template editor with some sophisticated features to help ensure that all windows you design have a consistent look.

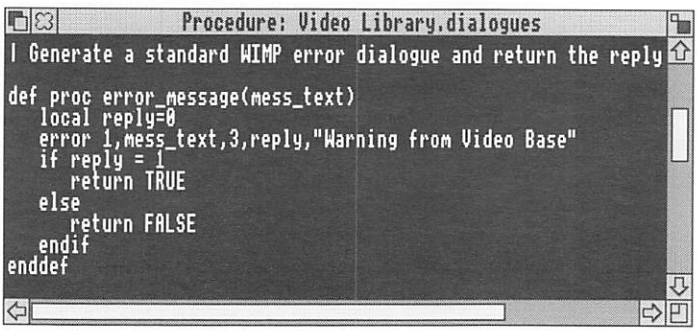
Constants



Constants are labels you set up in the SDE to represent an expression, string, or number such as commonly used complex values. For example you could define PI as 3.1412654 using the label PI and use PI instead of the number throughout your application. If you then discover your mistake and decide to make PI more accurate (3.141592654), the change will instantly reflect itself everywhere in your application. Constants are most helpful in data types and S programs for readability and consistency.

To allow S-Base to distinguish constants from variables, fields, and commands, you must give them upper case labels. When S-Base finds a constant in your application it substitutes the label with the value it represents.

Procedures



A procedure is a small chunk of S code which contains commands and

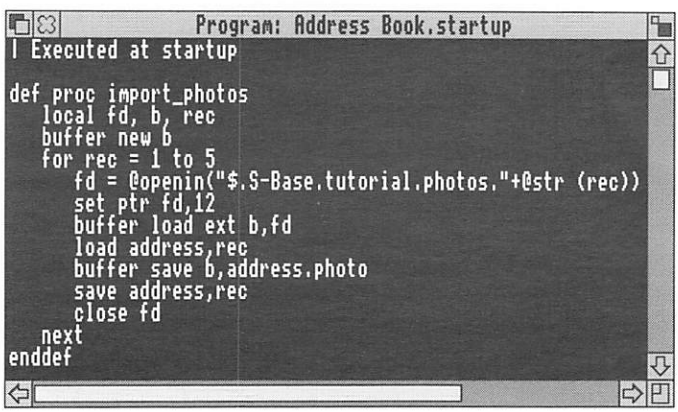
functions, and is used in the control and functionality of your application. It starts with the command **def proc** and ends with **enddef**. Procedures tend to consist of 10 to 30 lines of general purpose code designed to be used by all parts of your application.

The procedure shown above is called `info` (the name is always written straight after the **def proc**) and contains a small piece of code to display a card on the screen. Procedure names must be unique throughout the applicaton. Procedures can be passed values to process, and when they have finished their job they return back to where they were called from.

S-Base scans the procedure resource on start-up so that it knows what procedures are available. However, it does not execute any of them until asked to by your application. Procedures declared in the procedure resource are *globally* available for use at any time by any part of your application. Of course you can put **def proc..enddef** anywhere in your application to create a new procedure. These procedures declared in other resources are only available *locally* to that resource.

A procedure may return at any time to where it was called without having to execute the remaining instructions in that procedure. See Chapter 11, 'Control structures', for a detailed examination of procedures.

Programs



The *program* resource contains S commands and functions similar to one large procedure, but without the **def proc...enddef**. A program may in fact contain procedures that are declared local to that particular program (e.g.

`import_photos` in the example above).

Programs tend to be used for major operations and specialist code that is not used all the time. They are *run* rather than *called*, although this difference is fairly minor. Programs cannot be passed values nor can they return them. A program must be executed from start to finish before it may return to where it was run from.

S-Base executes two special programs to initiate and terminate execution of your application. A program called `startup` is run when your application starts, and allows you to take control of what happens from that point on. `windup` is the reverse and is run by S-Base when your application terminates. This is useful as it allows you to clean up any unsaved data if your application exits prematurely (crashes).

Programs may be organised into groups called *program lists*. These aid in structuring your application and keeping similar programs together. To reference a program stored in a program list S-Base uses the same system as the filer; objects inside other objects are separated by a dot. For example, a program labelled `hello`, stored inside a program list labelled `hello_list`, will be referenced as `hello_list.hello`.

Handlers

Handlers are similar to programs and procedures in that they contain S commands and functions. Their purpose is to handle events generated by the actions of the user. For example, if the user clicks on an 'OK' button a 'click' event is generated.

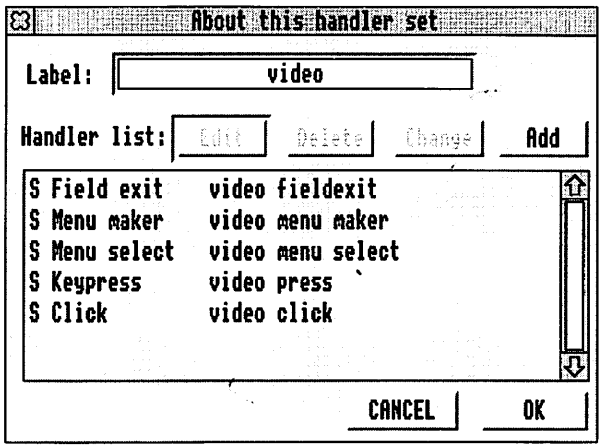
There are around 30 events generated in S-Base with a handler being called for each one. It is possible that a large application could well have hundreds of handlers. Advanced programmers should try to use the same handler for as many tasks as possible so as to reduce the number of handlers.

Handler sets

Handlers are grouped together into handler sets for ease of use with formats. A handler may belong to more than one handler set. For example, it is possible to design a single keypress handler which is used by all handler sets in the application.

A handler set can have one handler for each event type. It is not mandatory

to have a handler for each event type. The example below shows handlers for five different event types. If an event of a type not listed is generated, S-Base will handle it itself.



Handler sets are attached to format elements so that you can control what happens when the user clicks on an element, or presses a key inside an element, etc. For example, you might define a handler set for a 'Save' button, which saves the data the user has entered.

You can also attach a handler set to a format as a whole. Then elements which do not have their own handler set will default to the format's handler set. Attaching to individual format fields allows you to use numerous handler sets in the one format. Attaching to a whole format will handle all fields that do not have handlers defined, and for operations that apply to the entire format such as clicking on the Close box of a window.

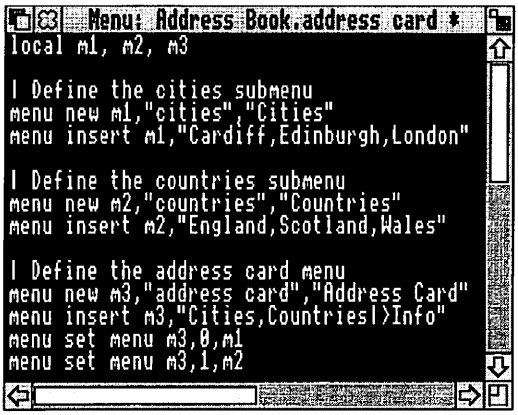
Menus

A menu resource is a set of S commands which creates a menu. Once created menus are kept in memory independently of the resource until they are disposed of.

The purpose of the menu resource is similar to that of the procedure resource. It is a convenient place to define the structure and content of menus that will be used throughout your entire application.

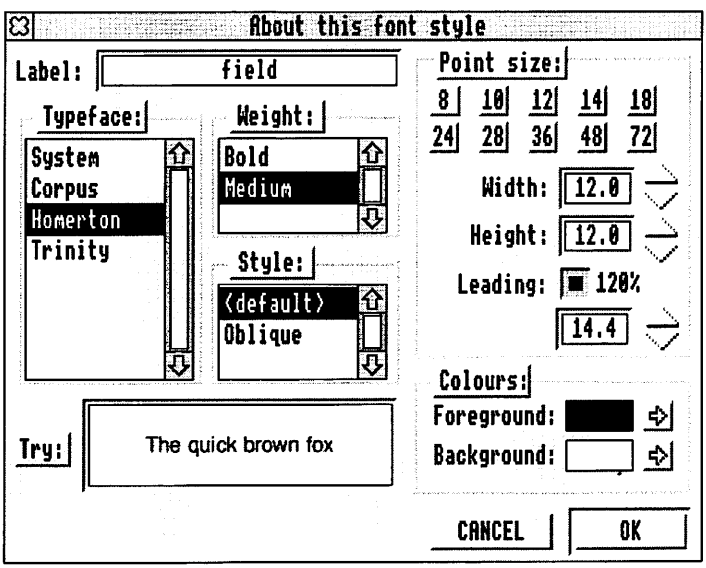
The menu resource is scanned at start-up so S-Base knows what is available. Other menus can be created by your application from programs

and procedures as they are needed; this will ensure you can generate menus that contain dynamic items. For example, the menu in the RISC OS filer window has a dynamic item: the second item of the menu changes each time you pop up the menu to display the file you have selected.


Menu: Address Book.address card *
local m1, m2, m3
| Define the cities submenu
menu new m1,"cities","Cities"
menu insert m1,"Cardiff,Edinburgh,London"
| Define the countries submenu
menu new m2,"countries","Countries"
menu insert m2,"England,Scotland,Wales"
| Define the address card menu
menu new m3,"address card","Address Card"
menu insert m3,"Cities,Countries|>Info"
menu set menu m3,0,m1
menu set menu m3,1,m2

S-Base menus support ticking and fading of menu items. Menus can be attached to *submenus* to create a large menu structure.

Font styles



This resource is used to describe how text looks in your application. Font styles are used in formats, allowing you to have text of any size, colour and typeface for cards, tables and reports.

The default font, System, is stored in the ROM of your machine and is therefore always available. If you have RISC OS 3, the font families Trinity, Homerton and Corpus are also ROM-resident and available at any time. Purchasing fonts from your local Acorn dealer will increase the range of typefaces available. Beware of sending applications containing lots of fancy fonts to people who don't own these fonts; they will not appear correctly and S-Base will have to use another font instead.

S-Base allows you to alter the width, height, leading (line spacing) and colours of any font style. Some fonts have different weights and styles, so S-Base also has provisions for selecting one of these.

Sprites

A *sprite* is a picture made up of coloured dots and can be of any size. It can contain two or more colours depending on the screen mode it has been created in.

Sprites can be created using the Paint application, which is found on the applications discs supplied with your computer, or art packages such as Revelation.

All sprites used within your application must be stored in the sprites resource. There is no limit to the number of sprites that may be stored in this resource except for that set by your computer (memory and disc space). Because of restrictions in the RISC OS operating system with regard to sprite names in icons, it is advisable to only use letters and numbers in labels.

The main use for sprites in your application is as icons in formats, templates and menus. Sprites are also useful in reports as logos and decorations.

Drawings

Drawings are similar in use and nature to sprites, though consist of lines not dots. Drawings may be scaled and altered without a loss of resolution.

Drawings are called 'Drawfiles' within RISC OS and are created with an

application such as Draw. S-Base cannot itself edit drawings – you must load them into Draw and edit them there.

The main use for drawings is in enhancing the appearance of reports and cards.

The next step

You should now have some idea of the components of an S-Base application. The next chapter shows you how to start up S-Base and begin your very own application.

5. The S-Base environment

All components of an S-Base application are divided into resources which interact and relate with each other. The SDE provides a fully integrated environment for designing, testing and running your applications.

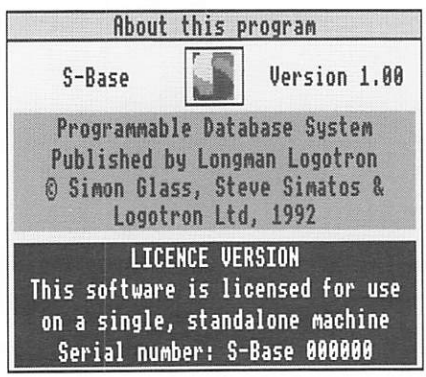
The SDE contains numerous dialogues and editors to allow you to edit resources. These editors handle all forms of resources from text to screen layouts.

The SDE will execute your application without the need for any separate software. Should an error occur in your application, the SDE will display an error message and show you the offending resource, highlighting the error.

To aid in debugging your application S-Base provides a *command line*. This facility allows you to display diagnostic information and execute simple commands and functions.

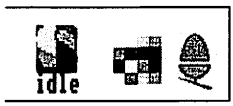
Keyboard short cuts throughout the SDE significantly speed up development time and make the S-Base environment easier to use. Once you understand the menus you should take time to learn some of the more common key short cuts.

Starting up



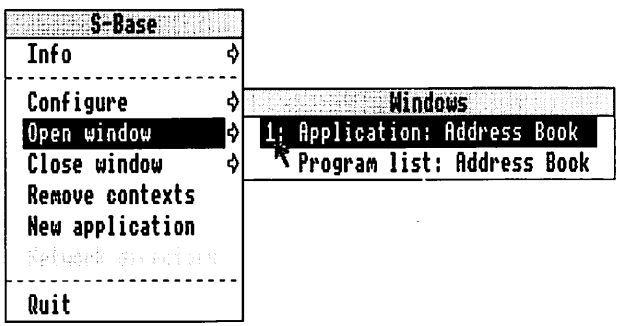
S-Base starts up in the same fashion as any standard RISC OS program, by double-clicking on the application. The first thing that will appear is the registration message. This will display the version of S-Base you are running, the copyright message and your serial number.

After S-Base has finished getting itself ready this will disappear and be replaced by an icon on the icon bar.



The S-Base icon displays the current state of S-Base and will be 'idle' at this point to indicate that it is not doing much. While S-Base is running your application this will change to 'running'.

Clicking Menu on the S-Base icon opens a menu containing items used in the operation of S-Base. The *Info* item displays the registration info dialogue presented on start up.



Configure allows you to check on and alter the operation of S-Base in various ways.

Open window and *Close window* display a pick list of windows. Most of the features here will only become clear when you learn about list viewers in the next chapter. Selecting a window from the open list will open that window. All currently open windows are displayed in the list above a separator with closed windows below. Selecting a window that is already open will cause it to move to the front. Selecting a window from the close list will close that window. If you hold down Shift while selecting the window its 'parent' will be opened.

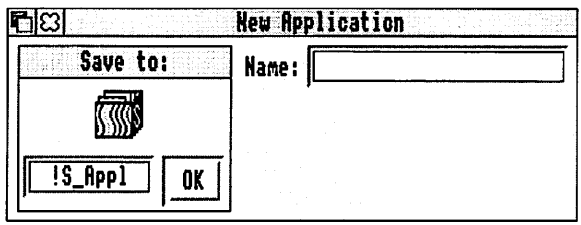
Remove contexts stops all applications from running and remove all windows created by the executing applications. This is equivalent to clicking in each application's resource list viewer and pressing Ctrl-F9. This option is useful when the application has cluttered the screen with cards or menus.

New application opens a dialogue for creating a new application. This operation is the same as just clicking Select on the S-Base icon and is covered in the next section.

Network directory is only supported in the network version of S-Base.

Quit will cause S-Base to shut down. This is the recommended way of exiting from an application. This option will prompt you if any data associated with your application is unsaved.

Creating a new application



It is now time to start your own application: an address book.

Select 'New application' from the icon bar menu or click Select on the S-Base icon. S-Base will open the new application dialogue and you can begin.

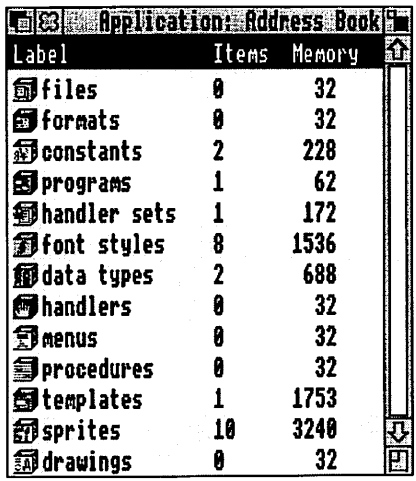
The filename inside the small 'Save to:' part will be used to create your application. This is limited to the maximum length of filenames on your computer's filing system (normally 10 characters, including the exclamation mark). The name may only contain valid characters recognised by your computer's filing system.

The application name to the right is the label used by S-Base to identify your application. This is a normal label and can be up to 31 characters. If the application name is not supplied, the filename is used instead.

Enter `!Address` into the filename box on the left and `Address Book` into the application name on the right. Now drag the application icon from the dialogue to the directory where you want to create your new application. S-Base will then create a new application for you in that directory and open the resource list viewer.

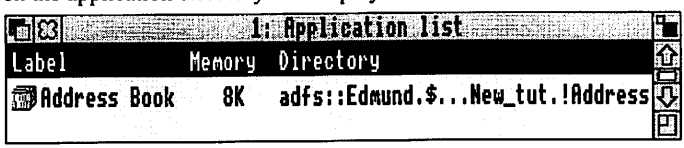
The resource list viewer displays a list of your application's resources with the number of items in each, and the memory they use. As you can see

below, your new application already contains a number of sample resources that you will find useful. These are provided to give you a foundation to build from. You will use this application later in this tutorial.



Managing applications

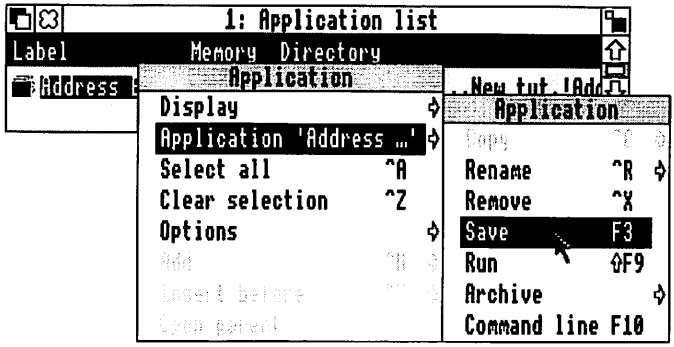
Clicking Adjust on the Close box of a window will open the parent of that window just like with any other RISC OS application. The parent of the list viewer shown above is the application list. This lists the applications currently loaded and whether they have been altered since last saved. Double-clicking on the directory path (shown here as 'adfs::Edmund...') will open the application directory and display its contents.



To get back your resource list double-click on 'Address Book'.

To remove an application from memory first select it by clicking on it then select 'Application–>Remove' from the menu. This does not delete the application from disc, it only removes it from memory. Try this with your address book application. To load it into S-Base double-click on the application icon in the filer window.

To save an application in the application list, select it and press F3 or select 'Application–>Save' from the menu.



An important keyboard short cut that you should remember is pressing Ctrl-F3 to save the application that is currently being edited. Try it now. You should see the disc light up, but since you have yet to add anything there isn't much for S-Base to save. You should get into the habit of pressing Ctrl-F3 regularly so that if something goes wrong you won't lose your work.

You can run your application by pressing Shift-F9 or selecting 'Select–>Run' from the application list menu. If it's already running, it will halt and restart.

The archive option is used to create an archive file for transmitting your application via modem or upgrading to a new version of S-Base. This option will be covered in more detail in the advanced sections of this manual.

The final item on the menu is the *command line*. This may be selected from the menu or by the keyboard short cut F10. The command line allows you to enter commands and functions and do some simple debugging. The command line always displays an `s:` prompt when waiting for you to enter a command. You are encouraged to type in the examples listed in this manual and to experiment yourself.

To follow

You should now understand the basics of loading and saving applications. The next chapter explains how to add resources to these applications.

6. Editing Resources

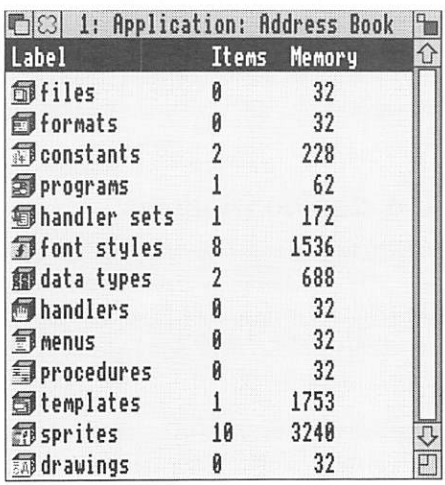
Resource editing and management is performed in an environment similar to the RISC OS filer with which you should be familiar. This chapter covers the basics of editing, manipulation and management of your resources, at the end of which you should be able to find your way around S-Base without much difficulty.

A more detailed description of particular resources will be included in later chapters as required.

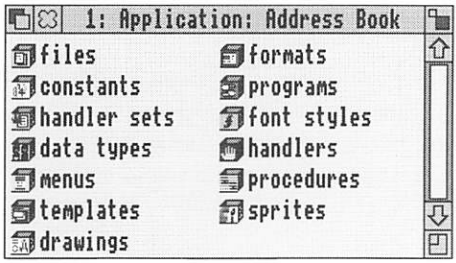
Viewers

Viewers are the most commonly used form of window for the display of resources in S-Base. They behave in the same way as the RISC OS filer viewers. If you are familiar with these then you are well on the way to mastering the S-Base environment.

The viewer you will become most familiar with is the resource list viewer. This lists the resources in your application, the number of items in each resource, and the memory used by each.



The resource list, like all other lists, can be displayed in *large icon*, *small icon* and *full info* formats. You can change the order using the Display menu. For example, selecting 'Display->Small' icons will remove all the extra information in the window and just display small icons.



You can also set the ordering used to display items. You can tell S-Base to leave the items in *physical* order (S-Base numbers all items internally), name, type or size order. By far the most useful view is full info with either physical or name order.

Window numbering

You will notice that all viewers have a number in their title bar. This number is assigned to the window when it is opened. You can use the function keys to bring an open window to the front. The window shown above is number 1, so you can bring it to the front by pressing F1. If you are using S-Base on a portable computer where mouse control is difficult you will find keyboard short cuts like this very useful.

The first 9 windows are numbered (1–9) in this way. Other windows don't get a number so you will have to use the icon bar 'Open' menu to open these.

Opening and closing windows

Double-clicking on a resource will open the viewer for that particular resource. Try the 'font styles' resource for example. Double-clicking on a resource object will open the editing dialogue for that resource. Try double-clicking on the 'field' font style. Click on 'Cancel' or press Esc to close the dialogue.

You can also open resources using the keyboard. First make sure that the cursor is in the relevant window (i.e. the window border appears highlighted). You may have to click in the window or press a function key to accomplish this. Then use the arrow keys to select the resource to be opened and press Return to open it.

You can close a viewer by pressing Esc or by clicking on the Close box on

the window. If you click with Adjust, S-Base will open the *parent* viewer. In fact, if you double-click on a resource with adjust S-Base will open that resource and close the one you came from. So you can navigate around an application using the Adjust button without leaving open windows lying about.

With dialogues, clicking Adjust on the Close box will open the parent viewer or dialogue in the same way. Sometimes you can navigate between dialogues without using a viewer, in this case the dialogue which opened the one being closed is reopened. If you hold down Shift while clicking Adjust in the Close box, the dialogue of the viewer is not closed, but the parent is opened.

All dialogues in S-Base have an 'OK' button which confirms the changes and a 'Cancel' button which cancels the changes. If you click on either icon with the Adjust button (rather than Select), the changes will be confirmed or cancelled as required, but the dialogue will not close. This is useful since you can easlily click with the Adjust button to see how the change will look without closing the dialogue each time.

Under RISC OS 3 you can *iconize* windows. Hold down Shift while clicking on the window's Close box. This reduces the window down to a small icon. The window is still intact but it is no longer cluttering up your screen. Double-click on the small icon to open the window again.

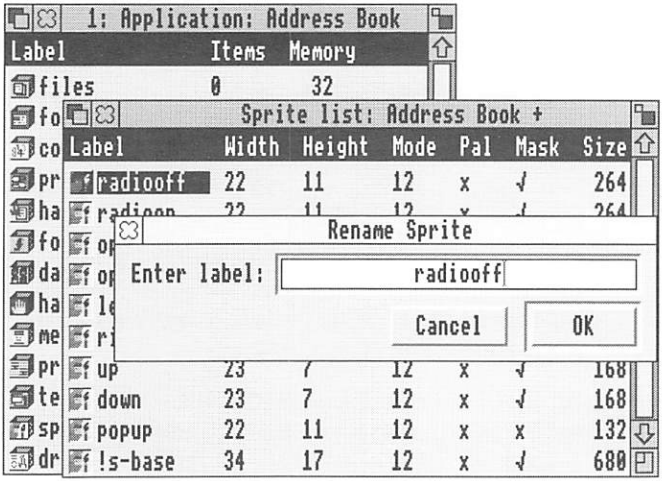
Selecting objects

Selecting objects works just like the RISC OS filer. Click Select on an object to select it. Click Adjust on an object to toggle that object between selected and unselected. Selections can also be created by dragging out a select box that encompasses all the objects in the selection.

You can also use the menu to select things. The menu offers two options: 'Select all' and 'Clear selection'. The keyboard short cuts for these are Ctrl-A and Ctrl-Z respectively.

Renaming resources

This works in the same way as the RISC OS filer. You can also rename an object by selecting it and pressing Ctrl-R.


1: Application: Address Book
Label Items Memory
files 0 32
Sprite list: Address Book +
Label Width Height Mode Pal Mask Size
radiooff 22 11 12 x √ 264
Rename Sprite
Enter label: radiooff
Cancel
OK
up 23 7 12 x √ 168
down 23 7 12 x √ 168
popup 22 11 12 x x 132
!s-base 34 17 12 x √ 680

Remember that two objects in the same viewer cannot share the same name. You must think of a unique name for each object.

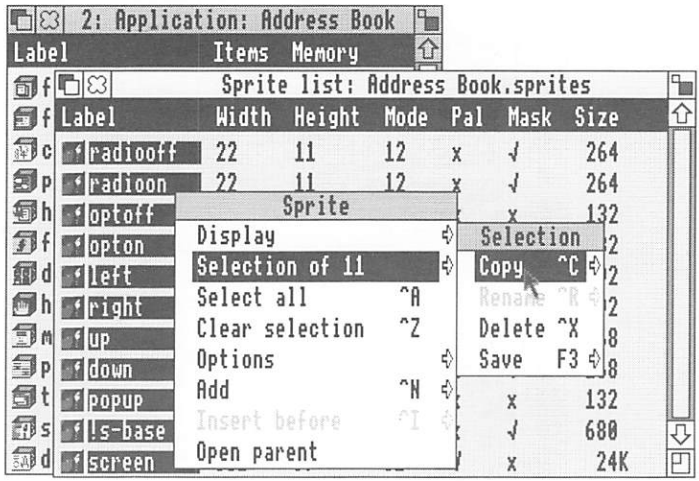
If you have used the keyboard short cut, then a rename dialogue will appear on the screen. The plus sign (+) in the sprite list viewer above indicates the window is being altered. Click 'OK' or press Return to rename the sprite. You can only rename one item at a time.

Copying resources

An object may be copied by selecting it and selecting Copy from the selection menu. You must enter a unique name for the new object.

The copy function will only copy objects that have been saved to the application. Objects in unsaved text editors or dialogues will not contain the unsaved changes. Copying does not affect the contents of the disc until the application has been saved to the disc with Ctrl-F3. Quitting from S-Base without saving the application will lose all copies made.

If you select more than one object, S-Base will not ask you for names for all the new objects to be created. Instead the new objects are numbered to ensure they are unique. For example, you can make a copy of all the sprites in the application by selecting them all then copying them.

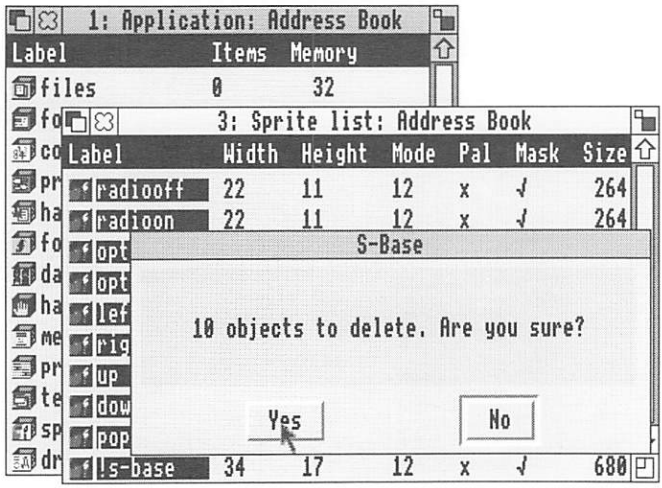


You may have to scroll the sprite list viewer down to see the new sprites. You will see that S-Base has made a copy of each of the sprites and placed a single digit (1) at the end of each label to make them unique. This automatic renaming occurs often where it is impractical to enter names for all the items being processed. You will also notice that S-Base now selects the new items. This is useful if you would like to delete or copy them again.

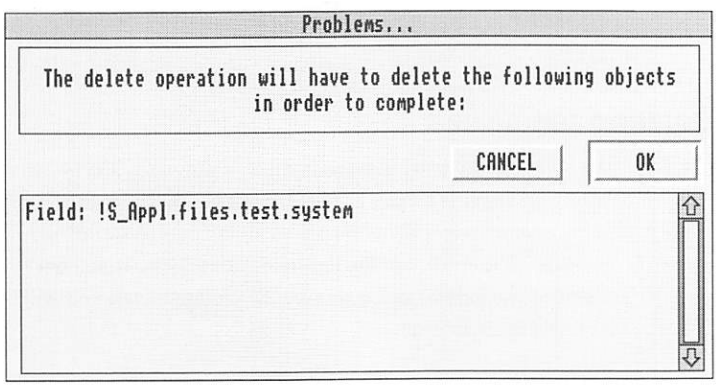
Deleting resources

To delete some items first select them, then select 'Delete' from the selection menu. You can also use the Ctrl-X keyboard short cut. After the copy operation has been completed all the copied sprites should still be selected. If not, select them before deleting them. If you have the *confirmed deletion* option set (see the Option menu) you will be asked to confirm the operation.

Since resources are interrelated you will sometimes find that the resources you want to delete are used by other resources. For example, if you have a file called `test` which has fields that use a data type called `system`, you will not be able to delete that data type. If you try, S-Base will tell you that the data type is in use by a field.


1: Application: Address Book
Label Items Memory
files 0 32
3: Sprite list: Address Book
Label Width Height Mode Pal Mask Size
radiooff 22 11 12 x √ 264
radioon 22 11 12 x √ 264
S-Base
10 objects to delete. Are you sure?
Yes
No
!s-base 34 17 12 x √ 680

The 'Problems' dialogue lists resources which use the resource you are trying to delete. You can cancel the operation or continue. If you continue, S-Base will automatically delete any resources which use the deleted resource. In this case S-Base would delete the `system` field of file `test`.



There are cases where this automatic deletion is not possible. If *this* field itself cannot be deleted because it is being used by something, S-Base will cancel the operation. Otherwise, taken to the extreme, automatic deletion would remove your entire application just to satisfy your request.

As with any operation you perform in S-Base the effect is permanent. The only way of recovering accidentally-deleted resources is to quit S-Base without saving and reload the application (obviously saving regularly

helps). You should keep the confirmed deletion option on until you are confident that you know what you are doing.

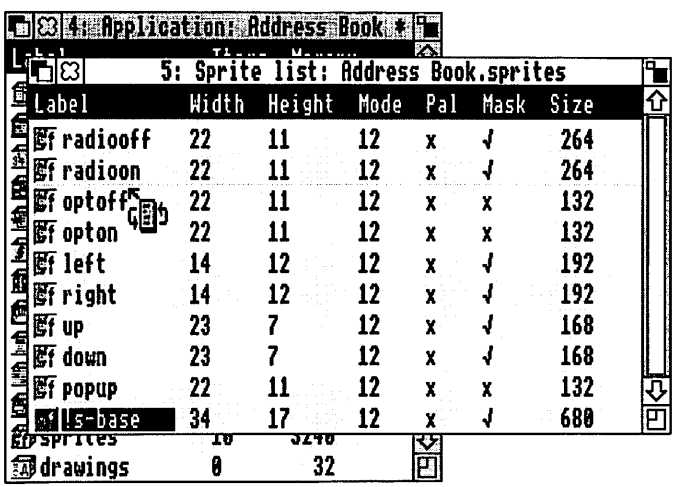
It is important to remember that S-Base edits applications in an integrated manner. When you open a viewer and delete something it disappears there and then. You do not have to confirm your changes each time you close a viewer. S-Base changes nothing on disc until you save your application.

Dragging

You can perform many tasks under RISC OS by dragging objects around. S-Base offers a number of ways of moving resources around, mostly performed with dragging. Just select the objects to be dragged and drag them to the required destination. S-Base provides auto-scrolling and auto-fronting (where the destination window is automatically brought to the front for you) as well as a destination cursor which shows where the objects will be placed. If you hold down Ctrl while dragging, S-Base will not auto-scroll or auto-front.

Pressing Esc while dragging objects will cancel the drag operation. This applies in all cases and is useful when you realise that you really don't want to complete the operation.

Reordering resources


4: Application: Address Book
5: Sprite list: Address Book.sprites
Label Width Height Mode Pal Mask Size
radiooff 22 11 12 x ✓ 264
radioon 22 11 12 x ✓ 264
optoff 22 11 12 x x 132
opton 22 11 12 x x 132
left 14 12 12 x ✓ 192
right 14 12 12 x ✓ 192
up 23 7 12 x ✓ 168
down 23 7 12 x ✓ 168
popup 22 11 12 x x 132
s-base 34 17 12 x ✓ 680
drawings 0 32

All resources are stored in the order they were created. This is called the *physical order*. You will usually edit resources in physical or name order.

With a viewer in physical order, you can change the physical position of an object by selecting it and dragging it to a new position. S-Base will show a coloured cursor indicating the new position for the object. You can only reorder one object at a time, and reordering is not available in the resource list viewer. You cannot reorder a window unless it is in physical order (for obvious reasons). Try moving the `!s-base` sprite above `optoff`.

Copying between windows

Selections or individual objects may be used in other objects by dragging them between viewers. For example, the `address` and `phone` fields in the `library` file of Video could be dragged into the viewer for the `customer` file.

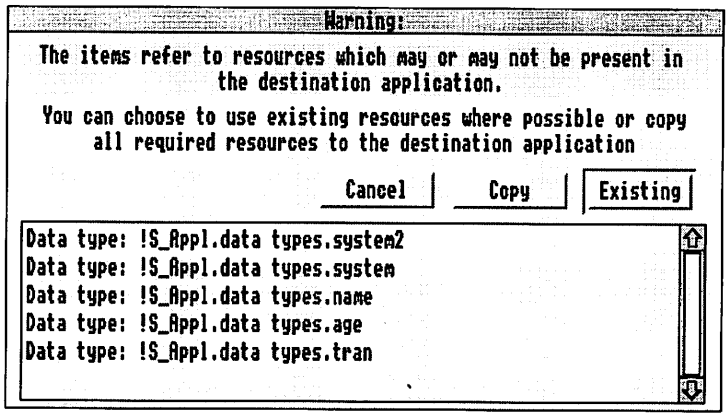
You can only copy between windows of the same type. For example, you cannot copy a sprite into the drawing list viewer.

If you press Shift as you start the drag a *move* is performed instead of a copy. This means that the original objects are automatically deleted when the copy is complete.

Copying between applications

Selections or individual objects may also be dragged between different S-Base applications. This is a convenient way of using resources created in another application without having to recreate them in your application. An example of this would be dragging the `customer` format from the Video system to a Stock Control system. Because a card display has quite a number of associated objects attached to it, S-Base would also copy the `customer` template, `customer` handler set and handlers and any font styles used. S-Base lists the resources that will need to be copied to the destination application.

You have the choice of copying the required resources from the source application or trying to use existing ones. For example, if you ask S-Base to copy a file which needs a data type called `system`, S-Base will allow you to copy the data type as well, or try to find an existing data type called `system` in the destination application. Note that if you use existing resources, they may not be what you expect. A `system` data type in one application is not necessarily the same as a `system` data type in another.


Warning:
The items refer to resources which may or may not be present in the destination application.
You can choose to use existing resources where possible or copy all required resources to the destination application
Cancel
Copy
Existing
Data type: !S_Appl.data types.system2
Data type: !S_Appl.data types.system
Data type: !S_Appl.data types.name
Data type: !S_Appl.data types.age
Data type: !S_Appl.data types.tran

Outside S-Base

It is useful to be able to use resources created in other RISC OS applications. S-Base allows you to drag sprites, templates, drawings, and text to or from packages such as Revelation ImagePro, Draw, Edit or from a filer viewer.

To load a resource into your application, save the file or selection from the source application or directory into any viewer in your application. S-Base automatically works out the correct viewer to place them in.

Try dragging a sprite file into your address book application (you can delete the sprites afterwards). The resources will be added after the existing resources.

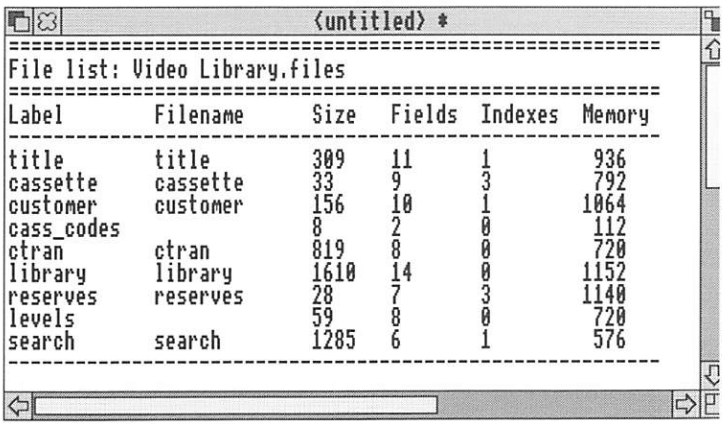
To save an S-Base resource to another application or directory, select one or more objects from the viewer of that resource and drag them to an application or directory. You can also press F3 or select 'Select->Save' from the menu to open a save dialogue for the current selection and this may also be dragged to an application or directory.

Note that some files can only hold one item. For example, you cannot select more than one drawing to save out, since drawing files can hold only one drawing. But it is easy to edit all the sprites in an application, just select them all and drag them into Paint.

Printing viewers

S-Base does not allow you to print the contents of a viewer directly, but you can save a text version of a viewer by dragging any objects that will open into a viewer when Double-clicked.

Drag the sprite resource from the resource list viewer into a text editor like Edit. This can also be dragged to a printer driver on the icon bar.


(untitled) *
File list: Video Library.files
Label Filename Size Fields Indexes Memory
title title 309 11 1 936
cassette cassette 33 9 3 792
customer customer 156 10 1 1064
cass_codes 8 2 0 112
ctran ctran 819 8 0 720
library library 1610 14 0 1152
reserves reserves 28 7 3 1140
levels 59 8 0 720
search search 1285 6 1 576

Creating resources

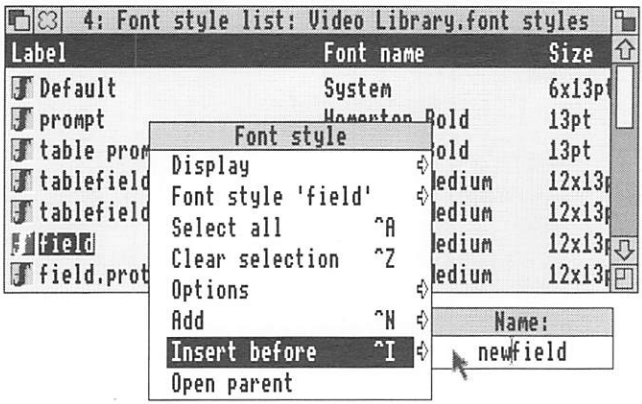
The creation and deletion of resources may be performed by keyboard short cuts or menus. A table of keyboard short cuts for resource editing is included at the end of this chapter.

To create a resource object click Menu in the appropriate viewer and select 'Add', or press Ctrl-N in the window. Enter a unique name for the object (up to 31 characters) and press Return. A blank object will be created and S-Base will open an appropriate dialogue box or editor for you to begin editing your resource. Adding an object in this way will always append it to the end of the list. You can reposition it later.

Try adding a new font style. Open the font style viewer and add a new font style called `test`. S-Base will add the style then open a dialogue allowing you to edit the style. This dialogue is explained later in this chapter. For now, just click 'OK'.

An alternative to adding objects is the *insert before* option. This inserts the created object at the position before the currently selected object. If more

than one object is selected it is positioned before the top-most selected object.



Delete the `test` font style you have just created.

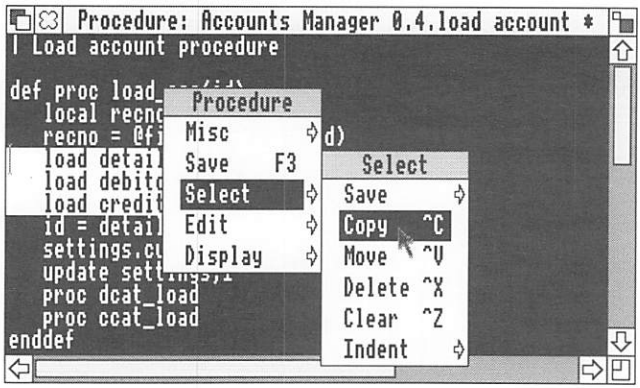
Editing

After a resource object has been created S-Base will automatically display it for editing. Alternatively, double-clicking on the resource object will load it into an appropriate editor or dialogue. This section will cover the editing of text, font styles and constants. All remaining resources will be covered in their individual sections later in this tutorial.

Note that all resource editing performed on an application will only affect the unsaved application in memory. The changes to the application can be permanently saved to disc by pressing Ctrl-F3. Exiting from S-Base without saving to disc will abandon all changes. S-Base will prompt you if you attempt this.

Changes to a resource in an editor will not be updated to memory until they are confirmed with 'OK' (for dialogues) or F3 (for text editors). An exception to this rule is made for changes to the format and template layout editors. Dragging operations in these editors are always updated throughout the application immediately.

Text editors



S-Base contains four *text* resources: programs, handlers, menus and procedures. These are edited in a text editor similar to Edit in functionality. However, S-Base saves its text into your application instead of a file. Clicking on the Close box will cancel the changes to the text. S-Base will prompt you before abandoning any changes.

Edit is described in detail in the RISC OS Applications Guide and that description is not repeated here. There are some extra functions such as the Edit->Insert menu option. These are described later in this tutorial when programs and handlers are introduced.

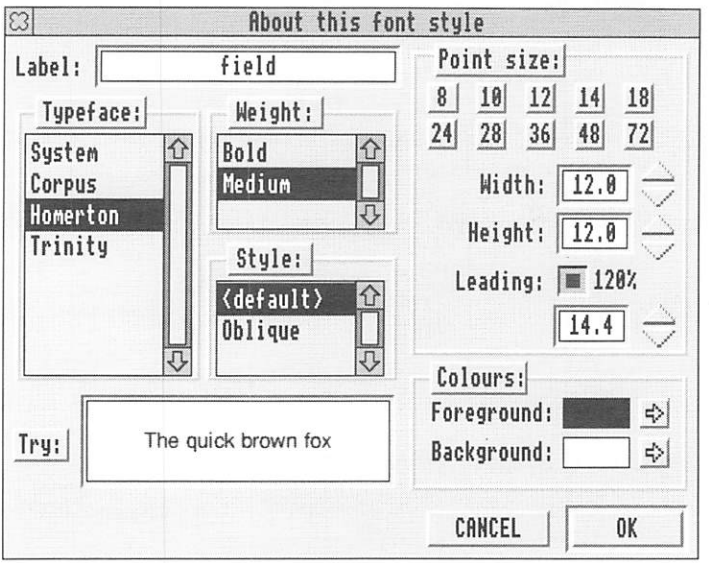
Font styles

The font style dialogue used in S-Base is similar to font selection dialogues used in other RISC OS applications. This dialogue allows you to set the characteristics of a font style and view a sample of it. These characteristics are: typeface, weight, style, width, height, leading, and foreground and background colour.

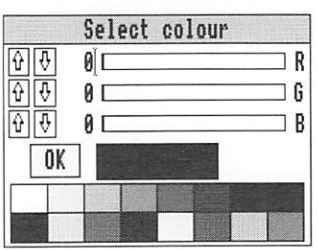
The lists of typefaces, weights and styles are displayed in scrollable selectors. The contents of these are derived from the fonts stored in your computer's font directories. Only one item from each selector may be selected at a time.

Font width and height are given in points and can be set by clicking on the point size icons, clicking on the up/down arrows, or typing in a point size. The leading is the spacing between successive lines of text. Selecting the

120% icon will automatically calculate a value 20% greater than the font height. Alternatively the leading may be entered as a numeric value or altered by clicking on the up/down arrows.



Foreground and background colours may be altered by clicking on the popup buttons next to them. This will open a colour selection dialogue. Click on a colour from the palette to select it and 'OK' to use it. The palette may be changed by dragging the RGB bars or clicking on the up/down arrows. This dialogue is similar to the one used in Draw to set the colour of objects. Note that S-Base will attempt to use the closest available colour to the one chosen, but since the desktop palette is controlled by the Palette Utility, S-Base cannot alter the palette just to display text in the right colour.

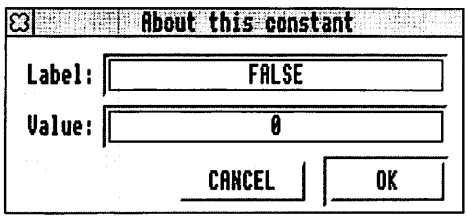


Once you have designed your font style it can viewed using the 'Try'

facility. Click on the 'Try' button to display a black on white image of the chosen font style.

Constants

The final dialogue covered in this section is for editing constants. This is the simplest of all the editing dialogues. It consists of a label field and corresponding value field. The label field will always be upper case as this is how S-Base detects the use of constants. *Constants* are used in data types and programs, and are covered later.



Keyboard short cut summary

A number of *keyboard short cuts* have been mentioned in this chapter. They are collected here for reference.

To use the keyboard short cuts available in this package, click in the window you would like to edit. This will highlight the window (give it the *input focus*), generally with a brighter-coloured title bar and scroll arrows. The following keyboard short cuts will perform operations accessed from S-Base menus.

Key	Action
F3	Save selection
Ctrl-F3	Save currently edited application
Ctrl-A	Select all
Ctrl-Z	Clear selection
Ctrl-C	Copy object
Ctrl-R	Rename object
Ctrl-X	Delete selected object(s)

Ctrl-N	Add object to end of list
Ctrl-I	Insert new object before currently selected object

In addition to these short cuts S-Base allows mouse click operations to be simulated by keyboard short cuts. These are as follows:

Key	Action
Esc	Cancel operation, or close window or dialogue.
F1	Click on 'OK'.
Left arrow	Select the item to the left of the currently selected item.
Right arrow	Select the item to the right of the currently selected item.
Up arrow	Select the item above the currently selected item.
Down arrow	Select the item below the currently selected item.
Return	Double-click on the current selected item.

7. Variables and Expressions

In this chapter the S programming language is introduced. S contains a number of commands and functions which you can use to give instructions to S-Base. The first step is to understand how variables and expressions work within S-Base.

What is a variable?

Variables should be considered as electronic boxes used to store one or more pieces of information. S-Base allows you to keep as many of these electronic 'notes' as you wish.

Each variable must be given a unique name consisting of a combination of any lower case letter or digit. This is so that you and the computer can distinguish one variable from another. For example, in an accounting system you might create a variable called `total_sales` which holds the total sales made during the month.

A variable name may only contain the characters a..z, 0..9 and '_', and must not start with a digit. Very long variable names will make your programs difficult to read and S-Base will take a long time to read them. The maximum length of a variable name is 249 characters. Try to use names that explain the variable's contents or purpose.

S-Base variables can store six types of information:

Integer	A value with no fractional part. This can be either 1, 2, 3 or 4 bytes long. The largest size can hold numbers from –2147483648 to 2147483647.
Float	A value with a fractional part. This can be either single (4 bytes) or double (8 bytes) precision. These store an approximation to real values e.g. 5.9 or –126.554. The precision refers to the accuracy with which the value is stored. Single precision allows 6 significant digits to be stored accurately (eg 123456, 0.000123456). Double precision allows 14.
String	A value of up to 254 alpha-numeric characters, e.g. "This is a string of 34 characters." String values must always be surrounded by quotes.

Date	A value representing a date accurate to one day e.g. 7/12/76. These can store a very wide range of years.
Time	A value representing a time accurate to one second, e.g. 7/12/76,15:34:21. These can store times for about 135 years starting from midnight on 1 January 1970. This may alter if a RISC OS 3 machine is configured for daylight-savings time.
Structure	A value representing a group of values. This will be introduced in Chapter 8.

Using variables

Load your address book application and press F10 to open the S-Base *command line*. This window allows you to issue commands to S-Base and see the results.

To assign a value to a variable an *expression* using the operator '=' is required. Try typing the following (remembering that the 's:' is the prompt given by the computer when ready for input)

```
s:a = 12345
s:print a
12345
```

This expression consists of two parts: a value to be evaluated on the right; and a variable to store the result in, on the left. The value component of an expression may be given as a single number or string, an expression, another variable, or the result of an S function, to name a few.

The **print** command above prints the value of an expression. In this case it prints the value contained in the variable a.

S-Base variables do not have a fixed type. You can change the type of data stored in a variable easily.

For example, try this:

```
s:a = 'A'
s:print a
65
s:a = 12345.6
s:print a
12345.60
```

```
s:a = "This is a string"
s:print a
This is a string
s:a = @timesn "7/12/85"
s:print a
07/12/85,00:00:00
```

As you can see from this short section of code the variable a has changed in type from an integer to a float, to a string, and finally to a time. The **@timesn** function converts a time in string form to internal form.

The example above demonstrates a feature provided for single character letters. S-Base will convert a character in single quotes (' ') automatically to its ASCII value and use it as an integer. The ASCII code is used by the computer to store characters. Each is stored as a number and you can see from the example above that 'A' is stored as 65.

Local variables

Local variables are variables that have been declared for use only in a particular object. Usually this is a program, procedure, handler or control structure. A local variable only exists while S-Base is executing the object it was declared in. For example, declaring a local variable in a procedure will mean the variable only exists while the procedure is being executed. When S-Base leaves the procedure the variable and its contents are destroyed.

To declare a local variable the command **local** is used. This command will contain a list of variables as its arguments, e.g. `local a, b, c`.

A **local** declaration must be given before the variable is used in the object. If another variable exists with the same name as the local variable, S-Base will copy it before replacing it with the local variable. When the local variable is destroyed S-Base will restore the previous variable and its contents.

Local variables should be used in all situations unless you specifically require a variable to be global to the entire application. In large applications, it is easy for a section of code to accidentally alter the value of a variable being used elsewhere. Often this will cause the application to behave unexpectedly and produce unusual results. This is known as a *side effect*. Making variables local will fix this problem.

Arrays

Arrays are another form of variable. An array is a group of values (generally all of the same type) with one label identifying the whole group. Array labels follow the same conventions used for variable labels, though they also use trailing square brackets for accessing array *elements* (members of the group).

Arrays are generally of a fixed type and size which must be given in advance, when they are declared. To declare an array the command **dim** is used.

For example, type:

```
s:dim i2 hello [100]
```

A **dim** command consists of the variable type (`i2` means 2 byte integer in this case), the label (`hello`), and the number of elements in the array (`100`). Each element in the array is identified by its unique *subscript* (0...99). The first subscript in the array `hello` would be `hello[0]`, the second `hello[1]`, the third `hello[2]`, etc. Note that 'subscript' and 'element' are often used interchangeably.

The **dim** command will automatically initialise the array to blank values ready for you to use. Every element in `hello` is now set to 0.

Arrays can be used with any other variable or file field (see later) of similar type though the result stored in the array will always be of the type defined in the **dim**. The following example attempts to set an element of `hello` to `300.1234`. S-Base cannot store the fractional part, so it ignores it.

```
s:a = 100.1234
s:hello[0] = 200
s:hello[3] = a + hello[0]
s:print hello[3]
300
```

The example array `hello` is described as being one-dimensional, by analogy with the 'line' of values `hello[0]`, `hello[1]`, ... it stores. This is why declaring an array is sometimes called 'dimensioning' and the **dim** command is used. S-Base supports arrays of up to three dimensions. Each dimension declared in an array must be contained in a separate set of square brackets.

For example,

```
s:dim i2 bye[3][4]
s:bye[0][0] = 1
s:print bye[0][0]
1
```

The two dimensional array `bye` is best demonstrated as a table:

`bye[0][0]`	`bye[0][1]`	`bye[0][2]`	`bye[0][3]`
`bye[1][0]`	`bye[1][1]`	`bye[1][2]`	`bye[1][3]`
`bye[2][0]`	`bye[2][1]`	`bye[2][2]`	`bye[2][3]`

Array subscripts must be greater than or equal to zero and less than the number of elements declared in the **dim**. Should this not be the case S-Base will produce an error message giving the correct range:

```
s:print bye[2][5]
Error: Subscript 5 out of range (0..3)
```

More advanced use of arrays will be covered in a later chapter of this tutorial.

Expressions and Operators

Expressions can range from the most simple assignments, as shown above, to long mathematical equations. To generate expressions other than assignments S contains a number of arithmetic operators for processing your data. The results of these expressions can be stored in a variable or compared with other data.

Each arithmetic operator is assigned a priority number. This determines the order that operators are evaluated in expressions and has a significant effect on the result. A table of operators supported by S and their priorities are listed below.

Priority	Symbol	Operator
1	@func	S-Base function
	–	Unary minus
	+	Unary plus
	!	Logical NOT
	( )	Brackets
	[ ]	Array brackets

2	^	Raise to the power
3	*	Multiplication
	/	Division
	//	Integer division
	%	Integer remainder
	<<	Shift left
	>>	Arithmetic shift right
	>>>	Logical shift right
4	$	Range check
	+	Addition
	-	Subtraction
5	=	Equal to
	<>	Not equal to
	>=	Greater than or equal to
	<=	Less than or equal to
	>	Greater than
	<	Less than
6	&&	Logical AND
	&	Bitwise AND
7	||	Logical OR
	~	Exclusive OR
	|	Bitwise OR

For example,

```
s:a = 12345.6 + 5^2 * 100
s:print a
14845.60
```

S-Base 'inserts its own brackets' according to operator priorities, so will evaluate this as `12345.6 + ((5^2) * 100)`.

Some operators may not be used in some contexts.

For example,

```
s:a = "This is a string" + 12345.6
Error: Type mismatch - string required
```

This expression produced a *type mismatch error* as it was not possible to add a float number to a string of characters.

When different types of data are evaluated against each other S-Base uses the first operand in the expression to determine the data type of the result. As you can see from the example above the result is expected to be of type string. This applies to all expressions except those between integers and float numbers. These are always evaluated to a float number as shown below:

```
s:a = 100 + 12345.6
s:print a
12445.60
```

The S operators can be classified into three groups according to their functionality: logical, bitwise and arithmetic. *Logical operators* compare the operands to produce a true (1) or false (0) value. S-Base treats any non-zero number as true and zero as false. It is possible to treat the results of all the operators as true or false. *Bitwise operators* compare binary versions of operands bit by bit to produce a numeric result while *arithmetic operators* calculate a numeric result.

Operators

'!' is the symbol used for *logical NOT*. This operator inverts the logical state of an operand.

For example,

```
s:a = 1
s:print !a
0
```

'&&' is the symbol used for *logical AND*. This operator compares two operands and if both are true (non-zero) then the result is true otherwise it is false (0).

For example,

```
s:print 3 && 2
1
s:print 3 && 0
0
```

'||' is the symbol used for *logical OR*. This operator compares two operands and if either is true (non-zero) then the result is true otherwise it is false (0).

For example,

```
s:print 3 || 0
1
s:print 0 || 0
0
```

'&' is the symbol used for *bitwise AND*. This operator compares each bit in two operands with a logical AND and produces a numeric result. Bitwise comparisons are made by converting each operand into binary and performing the comparison on each bit.

For example,

```
s:print 3 & 2
2
```

or

```
11 (3)
& 10 (2)
10 (2)
```

'|' is the symbol used for *bitwise OR*. This operator bitwise compares two operands with a logical OR and produces a numeric result.

For example,

```
s:print 3 | 2
3
```

or

```
11 (3)
| 10 (2)
11 (3)
```

'~' is the symbol used for *exclusive OR*. This operator bitwise compares two operands with an exclusive OR and produces a numeric result. An exclusive OR is the same as a bitwise OR except in the case of true exclusive-OR true which now equals false.

For example,

```
s:print 3 ~ 2
1
```

or

```
11 (3)
~ 10 (2)
01 (1)
```

This operator may be used to do a *bitwise NOT* by setting all bits to 1 in the second operand.

For example,

```
s:print 5 ~ 7
2
```

or

```
101 (5)
~ 111 (7)
010 (2)
```

'<<' is the symbol used for *shift left*. This operator converts the first

operand into a binary number and shifts it to the left by the number of places specified in the second operand.

For example,

```
s:print 3 << 2
12
```

The binary number 11 (3) is shifted left by 2 places equalling 1100 (12). The two bits on the right are filled with zeros.

'>>>' is the symbol used for *logical shift right*. This operator is functionally the same as shift left except that it is performed in the reverse direction.

For example,

```
s:print 12 >>> 2
3
```

The binary number 1100 (12) is shifted right by 2 places equalling 0011 (3). The empty places are filled with zeros.

'>>' is the symbol used for *arithmetic shift right*. This operator performs a shift right but maintains the value of the left-most bit (the 'sign bit'). This operator can safely be used on negative numbers.

For example,

```
s:print -12 >> 2
-3
```

An important point to note here is the way S-Base stores its integer variables. Unless otherwise stated they are stored as a signed 32 bit integer with the top bit being used for the sign bit. This means the logical shift operators will also shift the sign bit and affect the interpretation of the resulting number.

For example,

```
s:print -12 >>> 2
1073741821
```

The binary number 11111111111111111111111111110100 (−12) is shifted right by 2 places equalling 00111111111111111111111111111101 (1073741821). For this reason you should normally use >> on all signed numbers.

The final operator to discuss is '$' which is used for range checking. This operator allows you to check if a value in the second operand is within the specified set contained in the first operand, and returns true (1) or false (0).

For example,

```
s:print "1-5,7,11,12,20-100,13" $ 12
1
s:print "hello,goodbye" $ "hello"
1
```

The set is represented as a string and can use '–' to specify a range of numbers and ',' to specify individual values. Values are not required to be in any particular order and may also contain floating point values, and strings. All values must be of the same type.

Functions

S provides a large number of *functions*, many of which help when performing calculations. Function names are always lower case and are always prefixed with '@'. The functions available in S are listed in the S Reference, and introduced throughout this tutorial as required.

For example, you can get the square root of a number using **@sqr**.

```
s:print @sqr 16
4.00
```

Here the function **@sqr** is called with 16 as an argument. You may be accustomed to placing brackets around the argument, for example,

```
s:print @sqr (16)
4.00
```

However, S does not generally require this for functions which take only one argument. S-Base evaluates the first thing after the function and takes this as the value of the argument.

For example,

```
s:print @sqr 16 + 9
13.00
s:print @sqr (16 + 9)
5.00
```

You can see in the first example that S-Base has taken `16` as the argument to the function, and then added `9`. If you want S-Base to take `16 + 9` as

the argument, you must enclose the whole thing in brackets. S-Base works this way so that functions can be used within more complex expressions without gobbling up the rest of the expression as its argument:

```
s:print @sqr 16 + @sqr 25
9.00
```

Managing variables

S-Base provides the command **lvar** for you to get a list of active variables along with information on their type and current value.

For example,

```
s:clear
s:a = 100
s:b = 200
s:lvar
2 variables using 8 bytes (storage for 256)
Name                    Type      Value
-------------------------------------------
a                       Integer   100
b                       Integer   200
-------------------------------------------
```

S-Base uses a sophisticated memory management system that allows variables that have been defined to be removed and their memory given back to your computer's operating system. The command **clear** allows you to remove specified variables or all if no arguments are given.

For example,

```
s:clear a
s:lvar
1 variables using 4 bytes (storage for 256)
Name                    Type      Value
-------------------------------------------
b                       Integer   200
-------------------------------------------
```

8. Files and File fields

The structure of a file is defined using file fields. Each field is a unit of data to which you have assigned a data type. These file fields are grouped together to make records which are stored in your file.

This chapter describes how to edit the files resource, and differences between fields and ordinary variables. You will create a simple file for use in your Address Book application. More advanced use of files will be covered later.

Differences between fields and variables

Although file fields and variables are similar in nature and use, there are major differences in their implementation in S-Base.

The main difference between the two is the way they are stored and accessed. Variables are stored in memory and are lost when you exit from S-Base. They may be of any data type and are not fixed to that type in any way.

Variables may generally be declared or removed at will. They may contain digits in their labels. In general, variables are very flexible as their name suggests.

File fields are the reverse when it comes to flexibility. They must be defined in advance, in the files resource, along with their associated data type. As a result of this, a file field is fixed in type and can not be altered in any way. Provided a suitable data type has been defined there is no restriction to the structure of a file field's type.

File field labels are restricted to 'a'...'z' and '_', and may not contain digits. File fields may be stored in either memory, like variables, or on disc in a record.

If saved, fields are not lost when you exit S-Base and are useful for storing long term data. They may also filter data being assigned to them by use of minimum and maximum ranges defined in the data type.

Because of their fixed size and the manner in which S-Base implements them, you are always assured that there is sufficient memory to use a file field. This is not always the case with variables which do not have memory assigned to them until they are used.

File field labels are always shown prefixed by the file label and a dot. This is so that you and S-Base can work out which file the file field is used in. For example (don't try this):

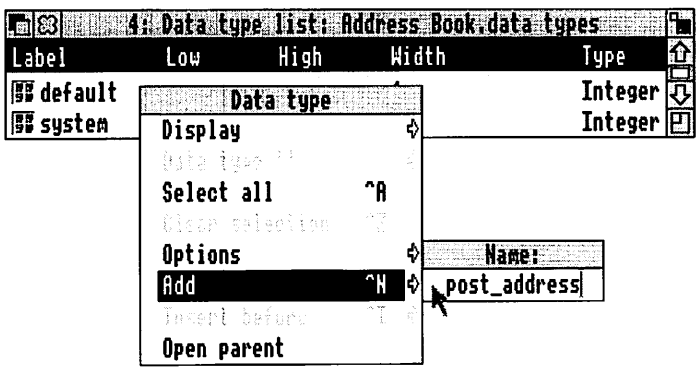
```
s:customer.total_issues = 10
s:print customer.total_issues + 3
13
s:customer.address[0] = "123a Acorn Road"
s:print customer.address[0]
123a Acorn Road
```

This example demonstrates adding `3` to the `total_issues` file field and storing an address in the `customer` file.

Data types

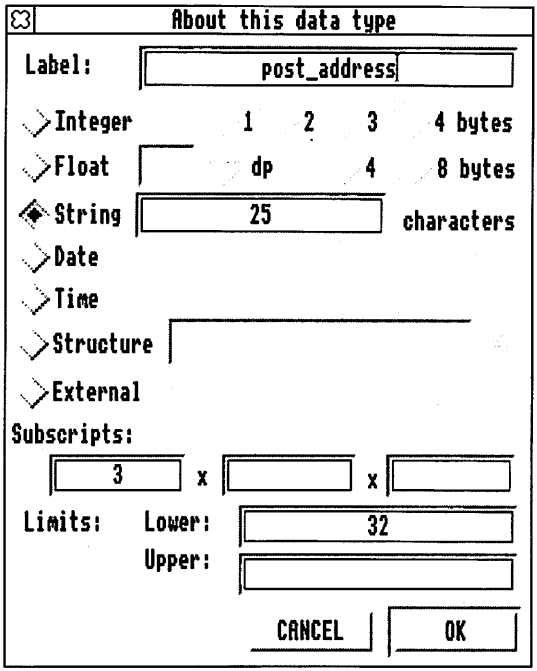
Data types are the building blocks for all files so before adding any files you will need to create some data types.

Close the command line window and double-click on the data type resource in the resource list viewer of your Address Book application. You will see the list contains two predefined data types used by S-Base. Now add a data type for a postal address.



Press Ctrl-N or select Add from the data type menu. Enter `post_address` and press Return. S-Base has now created a new data type and is displaying it for you in the data type dialogue.

The first field, as with all resources, is the label. This must be lower case, 31 characters or less in length, and may contain the characters 'a'...'z', '_', and space. You will see your label `post_address` has been entered here for you.



Below the label is a list of the basic types S-Base supports. They are the same as for those listed in variables though the lengths and accuracy must be defined now. There are two new types of data which you have not encountered yet: *structure* and *external*.

Structure allows your data type to contain the structure of an existing file. External is a handle of a data block in a *disc heap* where S-Base stores very large file fields. Both of these data types will be explain in the advanced files section of this tutorial and can be ignored for now.

By default S-Base has selected Integer 4 bytes as the data type. Change this by clicking on the radio button next to String (a short cut is to press Ctrl-S, since S is the first character of the 'String' prompt). You will notice the other fields are deselected and the characters field has become selected. Click there and enter `25`. This is the number of characters allowed in each line of the address.

The three subscript fields contain information on the number of dimensions and subscripts used for this data type. If a number is entered in the first subscript field it is implied that this is a one-dimensional array of that

many elements. If a number is then entered in the second subscript field it would imply a two-dimensional array, and so on.

Click into the first subscript field and enter `3`. There are now 3 lines in the address.

The final two fields in the dialogue are used to set the lower and upper limits of data values that this data type can contain. Fields for lower and upper data values are supplied with a blank value signifying no limitation. All limit values are specified as numbers. For strings, these numbers are converted into ASCII values.

Click into the 'lower' field and enter `32`.

This is the ASCII number for the space character. The user will not be able to enter anything that comes before 'space' in the ASCII table; this includes control characters that are not useful. Space is a common lower limit for string fields. No upper limit is set since the user may want to enter a copyright symbol © or perhaps some accented characters, e.g. á.

The `post_address` data type can be interpreted as: a data type named `post_address` which is an array of `3` strings each of `25` characters. The minimum allowable character that may be typed in is ASCII `32` (space) and no upper limit applies.

Now that you have finished editing the data type click 'OK' to confirm the changes. Clicking on 'Cancel' would destroy the changes.

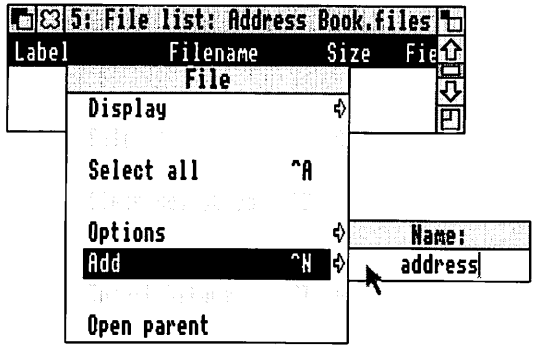
Try creating a data type called `name`. It should be the same as `post_address` though it will not contain any subscripts (leave out the `3`). You will use these two types in a file.

Creating files

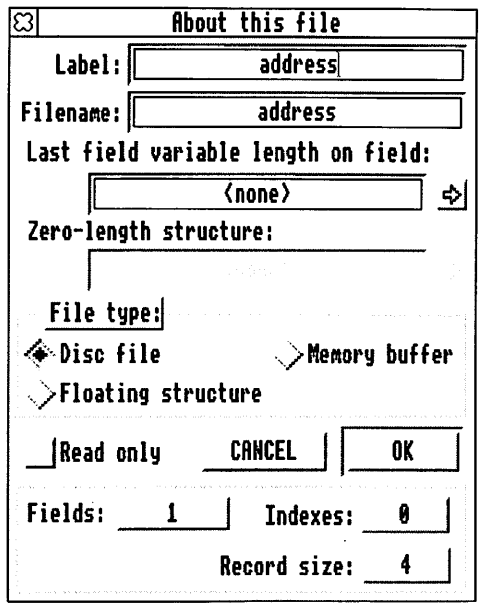
Files are created using the list viewers and dialogues similar to those described in the section above.

Close the Data Type List and double-click on the files resource in the resource list viewer to open a list of all the files in your application. This is empty at present so you must create an address file.

Press Ctrl-N or select 'Add' from the Files menu. Enter `address` and press Return.



You will see the `address` file appear in the files list and the file info dialogue will automatically open.



The first field in the file info dialogue contains the label of the file. This is the label S-Base uses to address all operations involving the database file and is only used inside the S-Base environment. It may contain the characters 'a'...'z' and '_', and must be 30 characters or less in length. All fields in this file must be prefixed with this label when used in formats and programs.

The *filename* field contains the name of the disc file in which you wish to store the data. This is limited to the maximum filename length of your computer's filing system, 10 characters usually, and is not used inside the S-Base environment. It is recommended, though not mandatory, that both the label and filename are similar if not the same. You will see the label and filename have already been entered for us.

The *variable length* and *zero length* fields are used for a special kind of file made up of variable length records. These can be ignored at present and will be discussed in a later chapter.

The *file type* may be classified as disc, memory or floating structure. *Disc files* are the type usually found in S-Base applications and can have any number of records which are loaded and saved from disc. S-Base always keeps one disc record in memory at all times so it is inadvisable to create file structures that use huge amounts of memory.

Memory files are the same as disc files except only the memory buffer exists. Thus there is only one record and it is never saved to disc. This record is therefore lost when you exit S-Base. Memory files are useful for storing temporary data where a variable is not convenient, or as temporary buffers (for advanced users) for use with your computer's operating system.

Floating structures are used in data types and allow you to include the entire structure of a file in another file. Floating structures don't even have a memory buffer, so you cannot store data in them. These can be ignored at present and will be discussed in a later chapter.

The *read only* button at the bottom allows you to ensure that the file can not be written to or updated. This is useful once the file has data. Files stored on CD–ROM and other read only media must be read only.

The remaining boxes contain information on the number of fields and indexes in the file and the number of bytes each record will occupy.

Click 'OK' or press F1 to confirm any changes.

Now that the file is ready you can create some fields in it. Double-click on the `address` file to open the file field list.

The file field list contains the field labels and the data types used by the file. The first field found in every disc file is the `system` field. This is a special field that was inserted by S-Base when the `address` file was

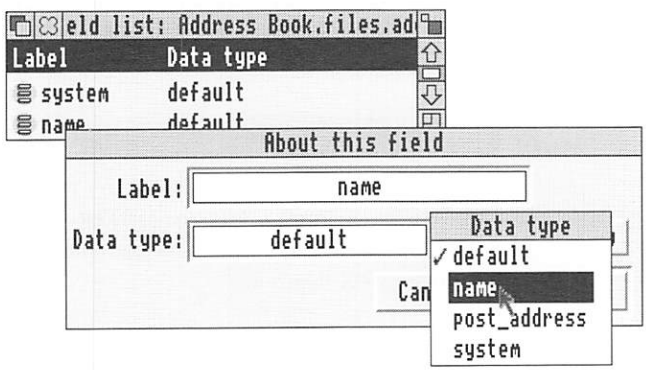
created. This is used by S-Base to store internal information about the record such as whether it has been deleted or not. Although this field is not mandatory for all file types it must be present in all disc files for them to function correctly, so do not delete the `system` field.

Press Ctrl-N or select 'Add' from the field menu. Enter `name` and press Return. S-Base will create the field and open the file field dialogue so you can edit the newly created field.

This dialogue contains only two fields, the label and data type. The label may contain the characters 'a'...'z' and '_', and must be 30 characters or less in length. Unlike the data type, it cannot contain spaces in the label. The label (`name`) has already been entered for us.

The data type label is selected from the data type resource, and can either be typed directly into the data type field or obtained by clicking on the menu pop up icon. S-Base has automatically assigned the default data type to the field so you will need to change it.

Click on the menu pop up icon and select `name` from the data types menu.



The Edit button to the right of the menu pop up icon will open the editing dialogue for the selected data type. To create a new data type from inside this dialogue enter the label in the data type field and click 'New'. S-Base will then create a new data type and present it for editing in a data type dialogue. This new data type will now appear on the data type list and is available to be selected from the current file field dialogue. This saves you trying to think of all the data types you will need before creating the file and file fields.

When you have finished editing the file field click 'OK' to confirm your

changes or 'Cancel' to abandon them (click 'OK' now).

Try adding an address file field to your file. Label it with `home_address` and use `post_address` as its data type. Once you have done this, the file is ready for use. Press Ctrl-F3 at this point to save your application to disc.

Using data types

You now know how to build a file structure by defining data types and adding fields which use those data types. The advantage with using data types is that if you decide that all addresses in your application should be four lines of 30 characters (rather than three of 25) then all you need do is change one data type. Provided all your address fields use that data type, then this is the only change you will need to make.

In fact you can go one step further. If you define a constant (say `ADDRESS_LINES`) which holds the number of lines you want in an address, you can use that constant whenever this needs to be known. You can put `ADDRESS_LINES` instead of `3` in the subscript part of the data type dialogue, and when you come to use the address fields in your program, you can use `ADDRESS_LINES` there too.

The record size

Now that you have added a file you can see the *record size*. If you look at the file list viewer you will see that the `address` file with the three fields you have created has a record size of 83 bytes. This means that for every record you add to the file, S-Base will require 83 more bytes of storage. This is not very much. In fact, S-Base could store almost 10000 such records on a single 800K floppy disc.

If you were to change the number of address lines from `3` to `5000` it would be a very different story. Each record would take up about 120K and you would only be able to fit about 6 records on a floppy disc. Also, since S-Base needs to store one record in memory, your application would require 120K more RAM to run.

The field order

In the example you have created three fields, with the `system` field first. The `system` field is created and used by S-Base and must always use the data type `system` (4 byte integer).

You can reorder fields in a file just by dragging them around in the list viewer, but you should never have any other field than `system` in the first position. Generally the order of file fields in a file does not affect you, since S-Base stores the data without you needing to worry about it. However, advanced users may want to extract the data directly, perhaps from outside S-Base. This is why S-Base allows you to specify the file field order.

The next step

You should now understand how to set up files and data types. This allows you to store data on the disc. The next step is to display this data on the screen, and this is covered in the next chapter.

9. Templates and Formats

The display and entry of data in all S-Base applications is performed using *templates* and *formats*.

Templates are blueprints that define the shape and function of windows. *Windows* are used to display information and to receive instructions from the user. Windows are displayed and handled by RISC OS under the control of S-Base. Your application need not be concerned with how to get a window on the screen, or how to display things in it. This is handled for you.

Formats are provided by S-Base for your application to enter and display its data. S-Base then passes the format information to RISC OS removing the need for you to worry about the complexities of the window environment.

Simply put, templates define the shape of the window and the icons within it; formats define the database information the user sees and what happens when the user does things.

This chapter will explain the differences between templates and formats and how they relate to each other. You will create a basic template and format for the address book application. More advanced use of templates and formats will be covered in later chapters.

Note for experienced users

If you have some programming experience in the WIMP environment, you will probably be familiar with templates already. You will most probably have used a tool such as FormEd to design your templates, and you will have noticed that almost all RISC OS applications of whatever size have a template file.

The template editor within S-Base provides all the facilities of FormEd and more. A template style system allows you to produce consistent look and function to your templates.

S-Base can generate a standard template file at any time. This process removes the style information, so such files are editable in FormEd and can be loaded into standard RISC OS applications. It is therefore possible to use S-Base as a template editor by creating a blank application with only your templates and sprites in it, and exporting a standard template file

when you finish editing.

Rather than using menus, the S-Base template editor uses dialogues. Also, some parts of the validation string have been separated out into separate options. For example, the icon border type is not stored in the validation string since the exact syntax depends on the RISC OS version that your application will be running on. Other than this, you should feel quite at home.

S-Base allows you to select several icons at once for editing. This is explained later in a discussion of advanced template editor features, but suffice to say that you can select a number of icons and change their border type, etc. in one operation.

What is a template?

A template basically consists of a window definition containing a number of *icons*. Icons are objects provided by the operating system for you to control your window. Icons may consist of text and sprites. They behave in a number of ways when you click on or move over them. The user can generally control your application by clicking on icons.

Some templates contain no icons and are used only to define the characteristics of your window. A window definition may contain numerous things like the contents of the *title bar* at the top, whether it has scroll bars, its size and shape, and colour. This is fairly confusing for the beginner. For this reason S-Base provides an easy-to-use template designer. The template designer allows you to quickly put together templates from pre-defined styles, thus giving a consistent and professional look.

Styles

When you look at well-written RISC OS applications you will note how they look and function similarly though produce completely different results. Acorn encourages software developers to use a common style in all their applications. This means using similar window colours, icons and functionality.

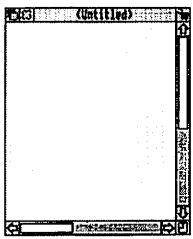
S-Base is a well-written application and encourages you to follow Acorn's standard. To help you keep a consistent style to your templates, a *style* system has been included in the S-Base template designer. This is similar to styles found in Desktop Publishing packages and allows you define how

a window or icon will look and behave. This style can then be applied to new or existing templates to give a consistent look.

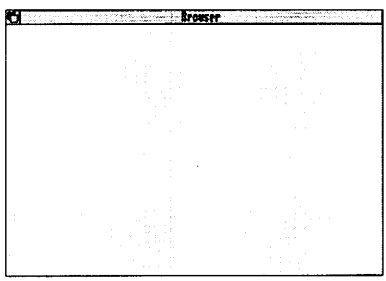
Note that S-Base is designed to run under any version of RISC OS, but has to adapt itself to its environment. An example of this is the 'look' of icons. These are drawn by RISC OS so they will appear differently depending on the version of RISC OS you are running. In general 3D effects are not available under RISC OS 2.

Window styles

Your address book application already contains the following window styles:



default – this is the default template. You will not generally use this style for anything, but you can use it as a basis for creating other styles or windows.



browser – this is a 'browser' window. It has no scroll bars or Close box and is useful as a record browser.

Icon styles

Your address book application already contains the following icon styles:

<Untitled>

default – this is the default icon. You will not generally use this style for anything, but it is a basic icon which you can modify.

action

action – this is a text icon used to indicate a course of action when clicked. The text in this icon is set by you.

box – this is a ridge used to surround groups of icons. It does not respond to mouse clicks, but makes windows look better.

box header

box header – this is a text icon used to label the group enclosed by boxes. It has no functional purpose and is solely cosmetic. You can set the header text yourself.

Cancel

cancel – this is used to cancel an operation.

default action

default action – this is used to indicate the default course of action when Return is pressed. The text in this icon is set by you. There is a separate OK style, so this style is only used for buttons which do not contain 'OK'.

left arrow – this represents decrement or move left.

OK

ok – this is used to confirm the current operation.

option option

option – this is a square sprite icon capable of taking two states: off and on. Clicking toggles between the two states. This allows the user to set an option on or off.

popup – this is a sprite icon representing a pop up window or menu.

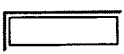
prompt:

prompt – this is a prompt icon used to label a field or icon. It does not respond to mouse clicks, but makes windows easier to understand. You can set the actual prompt text yourself.

radio – this is a diamond-shaped icon capable of taking two states: off and on. Clicking toggles between the two states. Several radio icons always appear together, the idea being that the user can only select one of the available options.

right arrow – this represents increment or move right.

sprite action – this allows the user to perform an action. You can set the actual sprite yourself. It should be simple and give a good idea of the action involved.



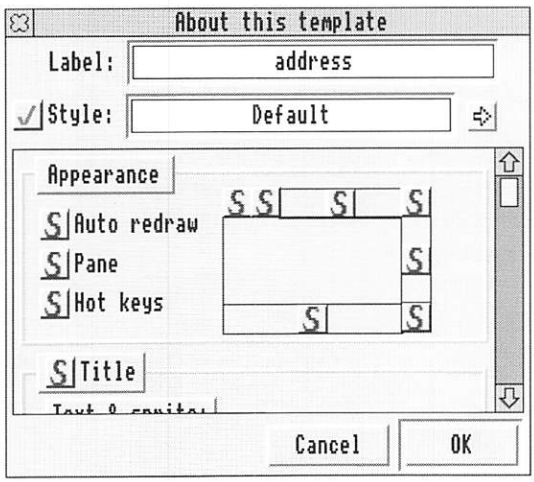
writeable – this is a text icon which allows the user to enter text into a buffer. This is similar to the text fields implemented by S-Base. This style should not be used in S-Base applications and is only provided for templates used in other RISC OS applications. Within S-Base you can use fields instead, and this is explained later.

Applying styles will be covered later in the following section. Creating and editing styles will be covered later in this tutorial.

Adding a template

Double-click on 'templates' in the resource list viewer of your address book application. This will open the template list which contains a single template called `default`. This template contains an example of each of the icon styles above. To look at this, select it and use 'Select–>Layout' from the menu. Close the window when you are finished.

Now add a template called `address`. S-Base will create the template and open the template info dialogue.



The top field is the template label followed below by the window style.

Click on the popup icon and select `browser` from the menu.

Each option in the template can take one of three states; each time you click on the option the state advances to the next one available.

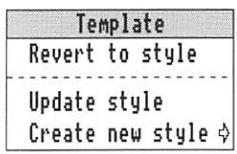
The style symbol means 'use the style'. This means that this template contains no value for the option, and the value should be obtained from the window style (`browser`). The icons around that window refer to the title bar and scroll bars, etc. At present, all of the icons are set to 'use the style', thus the window will have exactly the same appearance as the style dictates. The style you have selected is `browser`; This window style has no scroll bars and only a 'put behind' box in the title bar. Your newly created template will look the same.

If you click on one of the 'Style' icons S-Base will cycle through the three states. The tick means that you want the option 'on'. If you set the icon representing the Close box (second from the left) to 'on', then this template will have a Close box, overriding whatever the style's setting is.

The cross means that you want the option 'off'. If you set the icon representing the Close box to 'off', then this template will have no Close box, overriding whatever the style's setting is.

The bottom half of the dialogue contains a scrollable pane full of WIMP style options. For a description of each option and their rules of use you will need to consult the RISC OS Programmers Reference Manual.

Click Menu over the template info dialogue.



The template menu contains three items. *Revert to style* is used to set the icon and window options to the current style. S-Base will compare each option in the dialogue with the current style. If an option matches, its state icon is set to 'use the style'. If it does not the option remains unchanged. This facility is useful for applying S-Base icon and window styles to templates dragged in from FormEd.

Update style will update the current style to match the options in the dialogue. Using 'Revert to style' after this option will change all the state icons to 'use the style' since they will all match the style.

Create new style creates a new style containing the option settings from the dialogue.

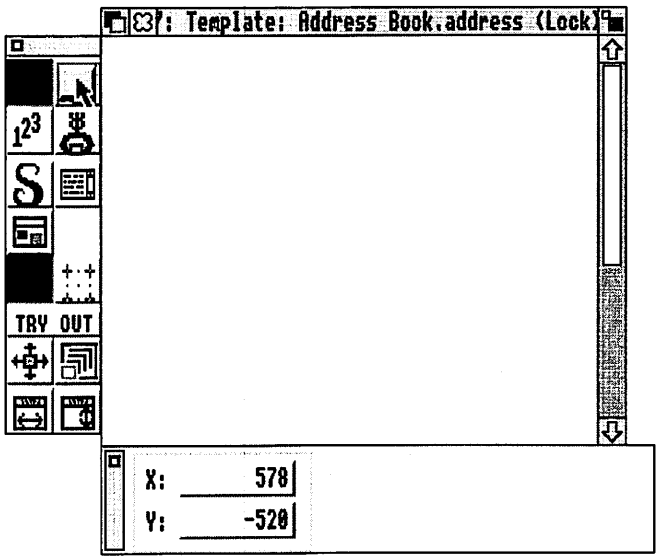
For now, leave all the settings as they are ('use style' for almost everything) and click OK.

Editing templates

To edit a template you can use the standard list viewer, but of course it is much easier to see how the template will actually look on screen. The template layout editor is provided for this purpose.

To edit your `address` template first select it then use 'Layout' from the

'Selection' menu or Shift-double-click on it. Either way, you will see a window with a *tool bar* down the left and an *info bar* at the bottom.



The tool bar is attached to the left hand side of the layout editor and contains a selection of tools used to create icons and manipulate your template.

The tool bar and info bar both have a small title bar at the top which you can use to drag the bar around. The small box in the title bar lets you close the bar.

Here is a list of the main tools in the set:

The selection tool

This tool allows you to select, move, resize and alter icons via a dialogue.

The creation tool

This tool allows you to create an icon by dragging out an area of the template.

Renumber icons

This allows you to renumber your icons when the template is finished. Within S-Base icons are seldom referred to by number, but this facility is often useful when you need to export templates to other applications. Renumbering is just another way of reordering your icons. The same can be achieved by dragging icons around in the icon list viewer.

Plunder icons

This allows you to 'plunder' icons from other applications. With this tool selected you can simply click on an icon anywhere on the desktop. S-Base will scan the icon and place it in your template. Note that some things that 'look' like icons are not icons at all. For example, the filer window does not use icons to represent files as you might expect; neither does S-Base use icons in list viewers.

The style list

Clicking on this will open the icon style list viewer. Ctrl-clicking on this will open the window style list viewer. You can then add and delete styles as you wish.

Alter template

Clicking on this allows you to alter the template. Ctrl-clicking on this allows you to alter the window style used by the template. This dialogue appeared when you created the template. Ctrl-double-click will open it from the template list viewer. While you are editing the template it is faster to just click here.

Template display options

This allows you to set the display options for the template layout editor.

Info bar control

This switches the info bar on and off.

Alter grid

This opens the grid dialogue. The grid allows you to accurately position icons.

Try out template

This creates a working version of your template for you to try out.

Move window

This allows you to move your test window around even if it does not have a title bar. Dragging anywhere in the work area will move the window.

Resize window

This allows you to resize your test window even if it does not have a size icon. Dragging anywhere in the work area will resize the window.

Scroll window horizontally

This allows you to scroll your test window horizontally even if it does not have a horizontal scroll bar. Dragging anywhere in the work area will scroll the window.

Scroll window vertically

This allows you to scroll your test window vertically even if it does not have a vertical scroll bar. Dragging anywhere in the work area will scroll the window.

The tool bar may be switched on or off from an option in the menu or by using the keyboard short cut Ctrl-F1.

The other pane attached to the template layout editor is the info bar. This displays the x and y coordinates (in OS units) of the mouse pointer and the name of the icon under the mouse pointer. The info bar may be switched on or off from options in the menu or tool bar.

OS Units

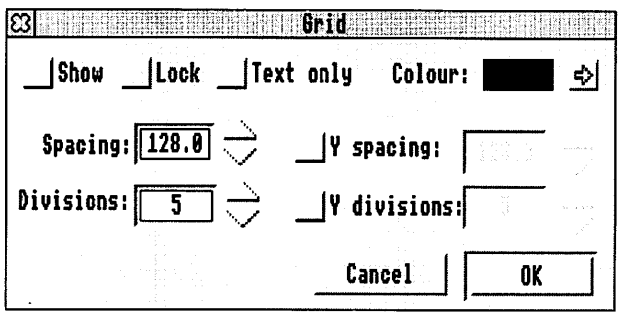
Templates and formats use Operating System units to measure distance and size. On an average monitor, 180 OS units corresponds to an inch (25 mm) across or down the screen.

The position of an icon is given in OS units as is the position of a format element (see later). Note that OS units do not correspond to a pixel or dot on the screen. In fact a pixel is generally 2 OS units wide and either 2 or 4 OS units high.

The grid

You can obtain a grid in the template layout editor to accurately position icons.

Click on the grid icon on the tool bar or press the keyboard shortcut Ctrl-G.



The grid dialogue contains settings for the grid attached to this template. The grid is useful for aligning icons in horizontal and vertical directions. Locking the grid will cause all icons to be created at grid intersections. There is also a facility to snap selected icons to the grid.

Click on 'show' and 'lock' to switch the grid on. Anything you do now will lock to the grid.

Spacing allows you to set the size of each grid square with divisions splitting each grid square into equal spacings. Spacing values are displayed in OS units and x and y dimensions may be set individually. A *text only* option is supplied to automatically adjust the grid settings to the same as the template system font.

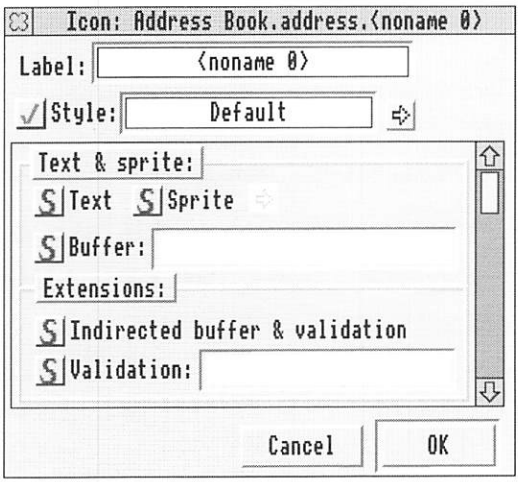
Click on the colour popup and select your favourite colour for the grid,

then click 'OK'. Click 'OK' in the Grid dialogue to confirm the changes.

Adding icons

The template will need some icons to control the `address` card.

Click on the creation tool on the tool bar (top right) and drag out a small box at the top left of the window. After creating an icon with default data S-Base will open the icon info dialogue. This dialogue is used to set the style of the icon.



S-Base has inserted a default label so change it to `left arrow`. You should always give sensible labels for icons as they are used later in the construction of formats. Next you will need to select a style for the icon.

Click on the style popup icon and select `left arrow` from the menu. When you have completed this click 'OK' to confirm the changes.

Some of the options in this dialogue will be explained later in this chapter. For a full description of the remaining options for icons consult the RISC OS Programmers' Reference Manual.

Ctrl-double-clicking on an icon in the template layout editor will open this dialogue for editing the style used by the icon. Shift-double-clicking on an icon in the template layout editor will open the font style dialogue for editing if a font style is used (this won't work for your arrow icon).

Try adding a right arrow using a label and style of `right arrow`. You should now have a left and right arrow in your template.

Positioning icons

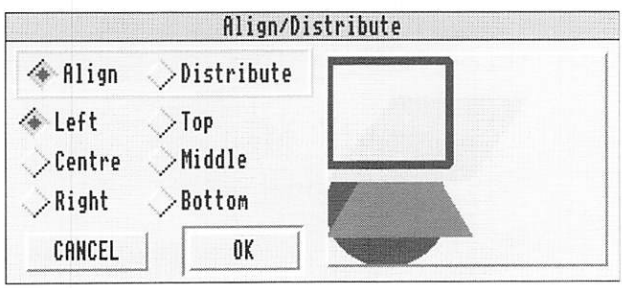
You should now position the icons at the top left corner of the template. First click on the selection tool (top left) and select both the icons by either using Select and Adjust or by dragging a box around them. You should see a coloured dashed border indicating that they are selected.

If an icon is obscured behind another object click with the Alt key depressed. This will toggle the selection through all the objects below the mouse pointer.

If you drag one of the icons to the correct position you will see both icons move. Position them in the top left of the template. You can drag any number of icons around in this manner.

Finer positioning is possible too. While the icons are still selected you can use the arrow keys to move them around. If you hold down Shift while pressing the arrow keys, the icons will move 10 times as far. If the grid is active they will move in grid units.

To position a selection of icons S-Base provides an Align/Distribute dialogue. This can be selected from an item in the 'Selection' menu or by pressing Ctrl-L.



The dialogue consists of two options and a selection of position buttons. The position buttons allow combinations of left/centre/right with top/middle/bottom. The drawing on the right side of the dialogue will show how the combination will effect a group of objects.

Each operation is performed in relation to a bounding box encompassing all of the icons. The align option will align each of the icons along a side of

the bounding box. The position combination will determine the alignment of the icons.

The distribute option will evenly distribute each of the icons across the combined bounding box. The position combination will determine the distribution of the icons.

Try adding a cancel icon using a label and style of `cancel`. You don't need to drag out a box – just click in the window with the creation tool. Use the `cancel` style and position this icon to the right of the arrow icons.

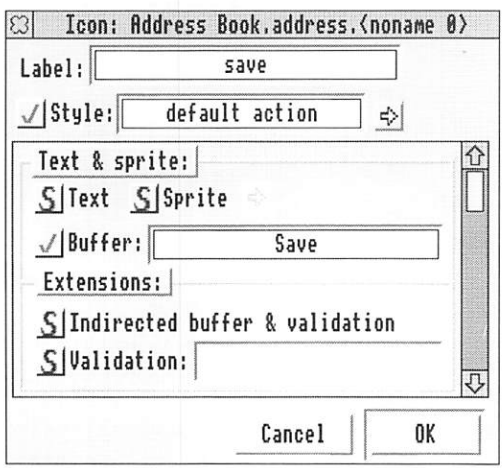
Altering icons

Double-clicking on an icon with the selection tool will open the icon info dialogue for that icon. You can view the options for the icon or edit it if you choose. Any changes will be reflected in the template layout editor when you click 'OK'. You can change the style of an icon as often as you like.

Style overrides

Sometimes the style is not quite what you want. For example, you may basically like the cancel button, but want the text to say `Save` instead. It is possible to override the style to achieve this.

Create a new icon to the right of the `cancel` icon (again just click – you don't need to drag out a box).



Enter `save` as the label and select `default action` as the style. Click on the 'use style' icon next to 'buffer' until it shows a tick. Now you have overridden the style buffer you can supply your own text for this icon. Enter `Save` next to the 'Buffer' prompt and click 'OK' to confirm your changes. You now have a default action icon containing the word `Save`.

Resizing icons

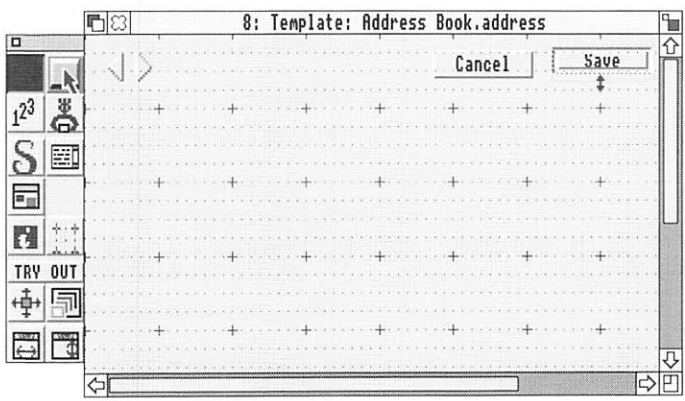
The size of an icon is determined by its bounding box. The bounding box can be much larger than the sprite or text it encloses and unless a border is provided, one will not be displayed. For example, the user could click inside the bounding box of the `left arrow` icon though be quite a distance from the `left arrow` sprite. This can be very confusing to the user and for this reason you should make sure the box closely matches the size of the sprite or text.

S-Base provides a 'fit' facility to automatically resize an icon's bounding box to fit the object it encloses. This can be selected from an item in the 'Selection' menu or by pressing Ctrl-F. Select all your icons using 'Select all' from the menu or Ctrl-A and 'fit' them by pressing Ctrl-F.

The icon styles provided with your new application have already been sized to their correct height and widths. Using these styles will ensure this problem never occurs.

However, text icons will often be too large or small for the text they display and will need to be resized.

Click on the grid tool and switch the grid lock off.



Select the save icon and drag its bounding box with Adjust to a larger or smaller vertical size. When you have finished switch the grid lock back on.

The direction in which an icon can be resized will depend on which area of the bounding box was dragged. For example, dragging the centre top will resize the top edge of the save icon in a vertical direction while dragging the top left corner will resize it in vertical and horizontal directions. The mouse pointer shape will tell you the side and direction you are resizing. Resizing with Adjust will also work with selections of any number of icons. Remember you can press Escape to cancel the drag.

Resizing templates

Click on the 'Try out' tool to display a working copy of your template.

You can drag this template around the screen to position it. When the template is used in your application, it will always open where you have positioned it with the 'Try out' option. If your window does not have a title (in this case it does) you can use the move tool to move it around.

You can resize the working window using the resize tool on the tool bar. Your template does not have a size icon so you will have to use the resize tool to resize it. The two scroll tools can be used also. Under RISC OS 3, you can scroll in both directions at once by using the Adjust button rather than Select.

All of these tools work by dragging within the template window, not on the window title bar or scroll bars.

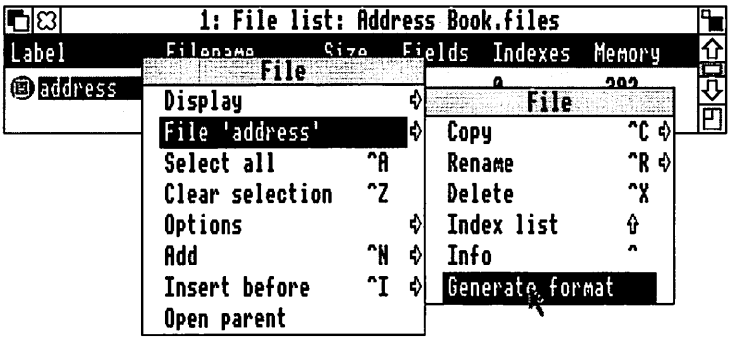
Use the resize tool to make your template large enough to hold the icons you have created plus a reasonable space below the icons for the format fields which you will create in the next section.

Clicking Menu over the working window will close it and set it to its new size and position. This is the same as clicking on the 'Try out' tool again.

Another way of positioning and resizing the template is to close the tool bar and info bar and drag the template layout editor around. This is the same function as dragging the test window around, updating the template position and size. Turn the info bar or tool bar (or both) back on from the menu or with Ctrl-F1 to return to normal editing mode.

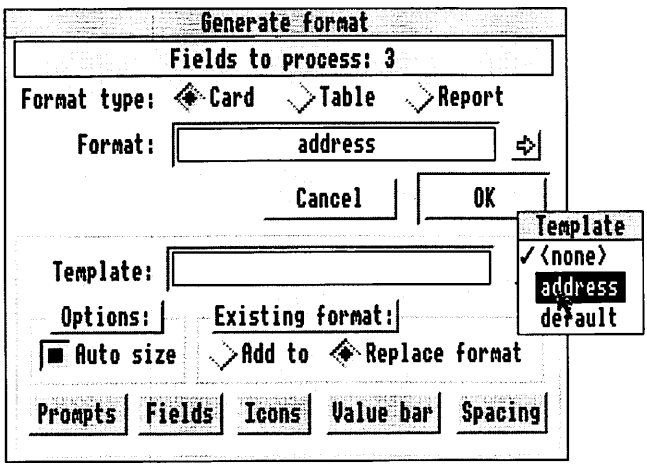
Creating formats

The final component to design for the address book is the *format*. You will use the format to tell S-Base what database information to put in the window. S-Base provides an automatic format generation facility to create formats at the click of a button.



Close your template and double-click on 'files' in the resource list to open the file list viewer. Click Menu over the `address` file and select 'Generate format' to begin generating a format for the `address` file. If you required a format to be generated for only a selection of file fields you could select fields from the file field list and select 'Generate format' from there.

Automatic format generation

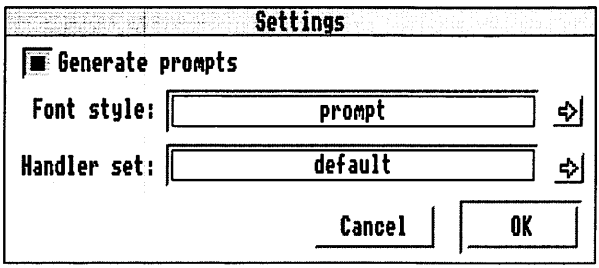


Automatic format generation (AFG) will generate formats for cards, tables and reports. You may add to or replace an existing format from a menu of formats attached to the format popup icon. If a format does not exist S-Base will automatically create it.

Enter `address` into the format label field. Click on the template popup icon and select the `address` template. This will be the template used for the format.

At the bottom of the dialogue there are a number of icons representing different components of a format. These are shaded light grey if not selected and dark grey if in use. The AFG dialogue only supports prompts, fields and icons. You will have to create other element types manually in the format layout editor.

Clicking on each of the icons at the bottom of the dialogue will open a settings dialogue requesting a font style and handler set for each component. Either of these fields may be deselected depending on the context in which this dialogue is used. When your format is created the data from these fields will be written into each element for you.



Click on the prompts icon from the AFG dialogue and select 'Generate prompts'.

The font style and handler set will select with default data. To change the current values click on the popup icons and select from the menus. The default settings are correct for the `address` format so click 'OK' (or press F1) to confirm the changes. Now try the same for fields, icons and the value bar using the default settings.

The *value bar* is a pane attached to the bottom of the format. It is used to display data from or associated with the field currently in use. The value bar may contain a larger font style than the rest of the format and is useful for users with impaired vision. It can also be used to display data related to

that being entered – for example, an expanded form of abbreviations or interactive help.

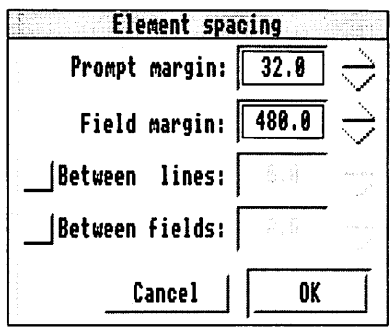
The final component of the format is the element spacing. Select the spacing icon to open the element spacing dialogue. There are four values making up the element spacing: the prompt margin, field margin, line spacing and field spacing.

The prompt margin is the distance in OS units from the left edge of the format window to the left edge of the prompts.

The field margin is the distance in OS units from the left edge of the format window to the left edge of the fields.

The line spacing is the vertical spacing between lines in a field. This is only used with elements generated from the same file field such as array elements. If no value is provided S-Base calculates a default based on the height of the font style used for the field elements.

The field spacing is the vertical spacing used between fields. If no value is provided S-Base calculates a default based on the height of the font style used for the field elements.



Element spacings may be entered from the keyboard or by using the up/down arrows on the right edge of the dialogue.

The default element spacings are correct for the format so click 'OK' to confirm them.

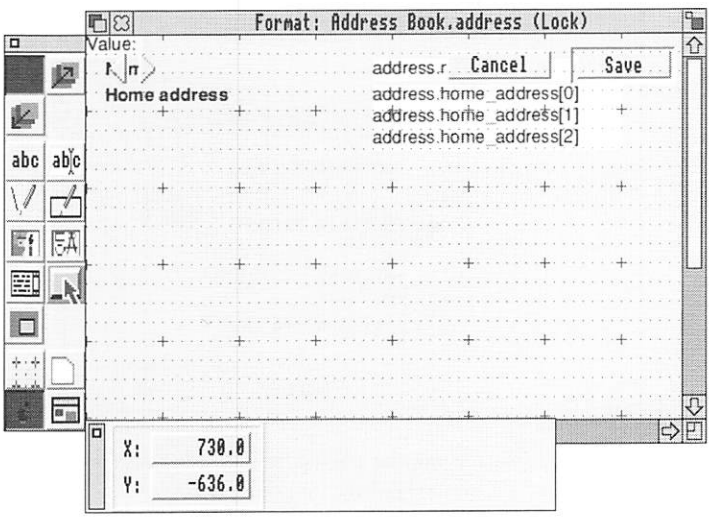
There is one option implemented for AFG of cards. *Auto size* automatically calculates the approximate width in OS units of each element field based on the font style and data type in use. If auto size is not set S-Base will fix

the field width to the expression string length for each field. The expression string is generated from the file and file field labels (e.g. `video.title`). If the size of a field is too small for the text within it, S-Base will vertically and horizontally scroll the field as the cursor moves through it.

The labels for each format element are generated in a number of ways. Format icons use the label of the template icon used to generate them and format fields use the label of the file field used to generate them. For this reason it is important to give meaningful names to all template icons and file fields. The remaining elements are given a label consisting of the object type followed by a number.

Click OK to generate the format.

S-Base may pause a while before displaying the new format in the format layout editor. You will notice the similarities between the template and format layout editors. Both contain a tool bar and info bar.



Here is a list of the main tools in the tool bar:

The selection tool

This tool allows you to select, move, resize and alter elements via a dialogue.

Bring to front

This allows you to bring an element to the front if it is obscured by other elements.

Put to back

This allows you to put an element to the back if it is obscuring other elements.

The prompt tool

This tool allows you to create a prompt element by dragging out an area of the format.

The field tool

This tool allows you to create a field element by dragging out an area of the format.

The line tool

This tool allows you to create a line element by dragging out an area of the format.

The rectangle tool

This tool allows you to create a rectangle element by dragging out an area of the format.

The sprite tool

This tool allows you to create a sprite element by dragging out an area of the format.

The drawing tool

This tool allows you to create a drawing element by dragging out an area of the format.

The selector tool

This tool allows you to create a selector element by dragging out

an area of the format.

The icon tool

This tool allows you to create an icon element by dragging out an area of the format.

The area tool

This tool allows you to create an area element by dragging out an area of the format.

Alter grid

This opens the grid dialogue. The grid allows you to accurately position elements.

Alter paper options

For reports this opens the paper options dialogue. The paper options set the dimensions of the page. For cards and tables this opens the template layout editor with the template for this format.

Info bar control

This switches the information window on and off.

Alter display options

This opens the display options dialogue. These options determine the colour each area will be displayed.

Remember the 'paper options' button. Clicking on it when editing a card format is a quick way of opening the template editor again.

Double-clicking on an element creation tool will allow you to alter the default data used by the tool to create an element. A description of how to edit the default data will be given later in this section.

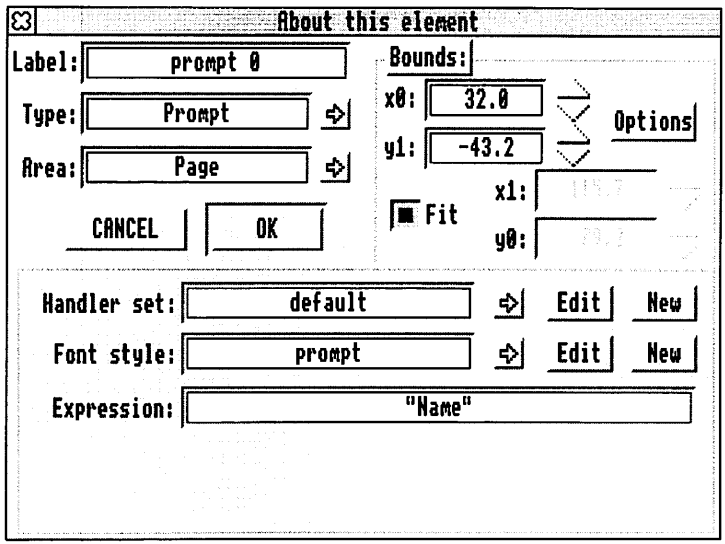
Ctrl-double-clicking on an element will edit the handler set used by the element. Shift-double-clicking on an element will edit the font style used by the element. Ctrl-shift-double-clicking on an element will edit the S Click handler used by the element.

S-Base has created a basic format and saved about 10 minutes work, though you will need to do a bit of repositioning. The selection, dragging and resizing rules applying to the template layout editor also apply here.

Select the prompts and drag them to a suitable position (or use the arrow keys). Then select the fields and position them next to the prompts. The grid will ensure they are aligned correctly. Be careful not to accidentally drag the icon elements (which will alter the template automatically). If you do, try to drag them back into position, or click on the paper options icon on the tool bar to open the template and fix the positions again.

You will notice the value bar element positioned in the top left corner of the format. This may be positioned anywhere in the window as S-Base always attaches it to the bottom of the format when it is used.

Double-click on the name prompt. This will open the element info dialogue for this element. This allows you to edit the element. Any changes made will be reflected in the format layout editor when 'OK' is clicked.



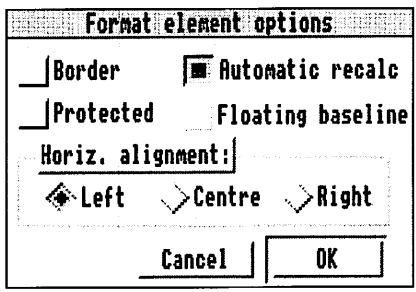
The element info dialogue consists of two parts. The top half contains details common to all types of elements: the type, bounds (position) and area type. There is also an options popup to set various options for an element. The bottom half contains information relating to the particular element type selected.

The bounding information describes the element's bounding box. This is given in OS units and relative to the window origin at the top left of the window. You will note that the y-coordinates are negative. This is because they are below the window origin. The window origin must be at the top of the window because the window can extend downwards to any size.

The bounding coordinates have been grouped together on the dialogue into top left (x0,y1) and bottom right (x1,y0). These may be altered by clicking the up or down arrows, or entering a value. They are stored accurate to $^1/_{10}$ of an OS unit. S-Base rounds these to the nearest pixel when displaying them.

Clicking the fit button will automatically alter the coordinates to fit the element data. The length of the expression string at the bottom of the dialogue is used for fitting fields and prompts. Icons are fitted to cover the corresponding icon in the template. Other element types cannot be fitted. When fit is selected only the top left corner of an element may be specified — the bottom right corner is calculated automatically. Fit will automatically deselect when the element is resized in the format layout editor: for example, dragging out the corner of a element with the mouse pointer will turn 'fit' off.

Click on the 'Options' icon to view the options for the `name` prompt.



The element options dialogue mainly controls the appearance of the element. Horizontal alignment specifies whether the element is left or right justified, or centred within the bounding box. The border option places a border around the field.

Protected specifies whether an element can be selected or not. Protected elements are 'greyed out' and cannot be altered or clicked on by the user. Protected fields are drawn with a different font style from non-protected

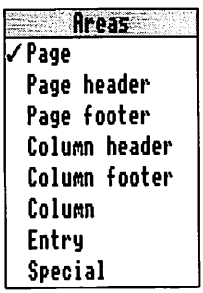
fields. S-Base appends '.protect' to the original font style and uses that style instead. For example, a field element with font style 'field' will use the font style 'field.protect' if it is protected. This gives you complete control of how to 'grey out' text fields yet keep them readable. If no protect font is found S-Base will use the existing font.

Floating baseline is only used in reports and allows S-Base to expand the baseline of a field if the bounding box is too small to display a long piece of text.

S-Base maintains a data entry buffer for each field element in a format. When the format is created the buffer is initially set up by evaluating the expression in the element. Automatic recalc allows you to control whether or not the element's expression is updated as the user types. If it is set on, S-Base will automatically assign the user's input into the expression.

Sometimes it is desirable to manually process the user's input. For example, you may want to use the third `home_address` field exclusively for entering an abbreviation for the name of the country where the addressee lives. By asking S-Base for the contents of the buffer (what the user has typed) your application can convert it into a full name then store it in the element's expression. The expression can then be displayed in the value bar while not overwriting the abbreviation in the buffer. The user will now see the abbreviation they have entered as well as the expanded name that has been selected. When the user exits the field the contents of the buffer can be updated with the full name in the expression. Automatic recalc must be switched off to stop S-Base from interfering with the field while you are processing it. An example of this is given in the find option in Video.

The final item in the common data is the area. Areas are selected from a popup menu of eight items.



The area describes which part of the format an element belongs to. Different areas tend to be used by different types of formats. For example, cards use page and special while reports use all the areas. The areas for tables and reports will be discussed in their chapters later in this tutorial.

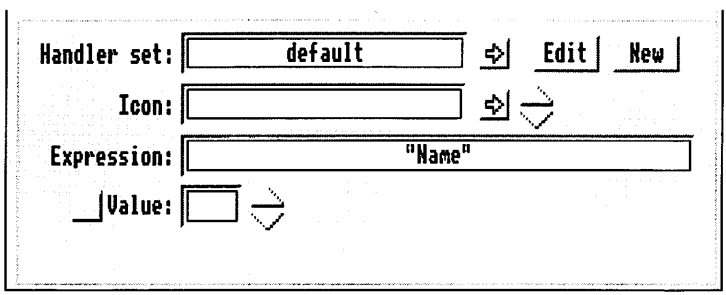
The area type for all elements in a card is set to page since cards do not contain headers and footers etc. The exception to this rule is the value bar which must be placed in the special area. Attached tables and user-drawn graphics also use the special area but these will be discussed later in this tutorial.

The area types for the format you have created are automatically set by AFG and should not need to be altered.

Element types

A popup menu is provided for you to select the type of an element. This will be already set by AFG and can usually be ignored. The second half of the element info dialogue is determined by the element type chosen. The name element is a prompt and requires a handler set, font style and expression to be entered. These can either be entered or selected from popup menus. Field elements are the same as prompts and require a handler set, font style and expression.

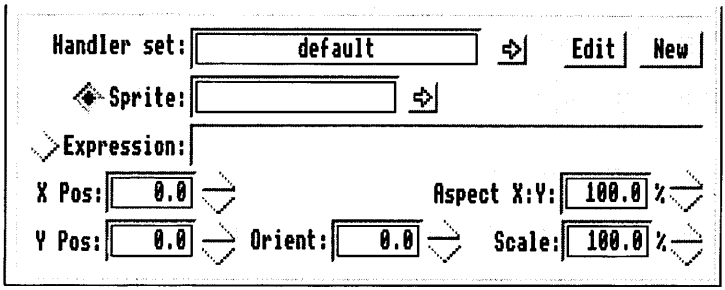
Selecting 'Icon' from the type popup will change the bottom half of the dialogue to accept icon information. Try this now.



Icon elements require a handler set, icon, expression and value. The icon must exist in the format's template. You can click on the popup to get a list of icons available. The expression is not used for a simple 'action click' but it contains a variable for use with radio icons. This is explained later in this tutorial.

The value field is used for groups of radio icons where only one may be selected at a time. When this element is selected the value is assigned to the expression, the icon is selected, and all others in the group deselected.

Select 'Sprite' from the type popup.



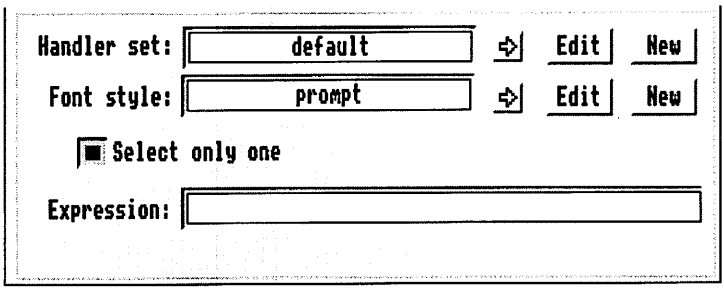
Sprite elements require a handler set, sprite name or expression, orientation (rotation), scaling and aspect ratios, and x and y position. The sprite name is selected from a menu of sprites found in the sprite resource. If an expression is used S-Base will evaluate it to display the sprite. This is only used in conjunction with external file fields and will be covered in a later chapter of this tutorial.

Orientation is given in degrees with 0 being the top of the format. Scaling is given as a percentage and the aspect ratio expressed as x as a percentage of y (e.g. 50% means the sprite width is half its height). The x and y positions are given relative to the origin of the element's bounding box.

If you give a scale of 0%, S-Base will automatically scale a sprite to fit the size of its bounding box. If the sprite is smaller than the bounding box it will be scaled up. The reverse applies if it is larger than the bounding box.

Drawing elements require the same data as sprite elements. Drawings are taken from a menu of drawings found in the drawing resource. If an expression is used S-Base will evaluate it to display the drawing. This is only used in conjunction with external file fields and will be covered in a later chapter of this tutorial.

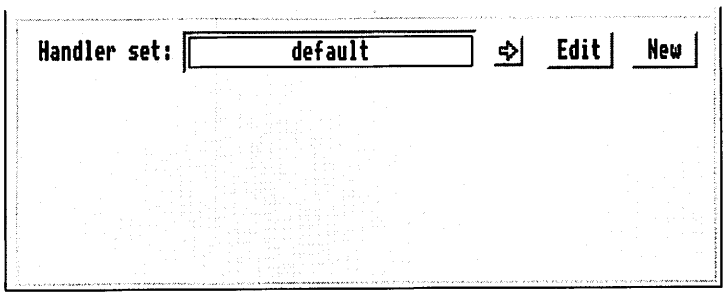
Select 'Selector' from the type popup.



If 'Select only one' is on, a variable or file field must be provided in 'expression' to hold the currently selected item number. If more than one item can be selected S-Base provides functions to out work which items are selected.

Select 'Area' from the type popup.

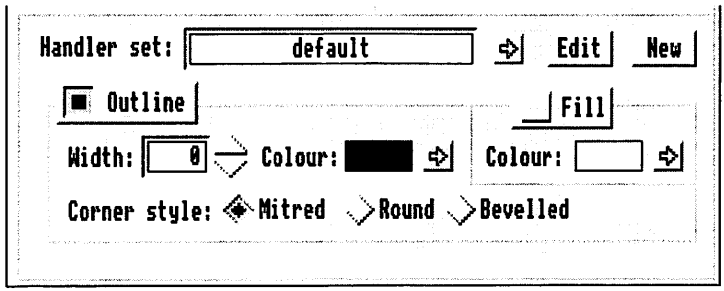
Area elements only require a handler set to be entered. The use of area elements will be discussed later in the table and report chapters.



Select 'Rectangle' from the type popup.

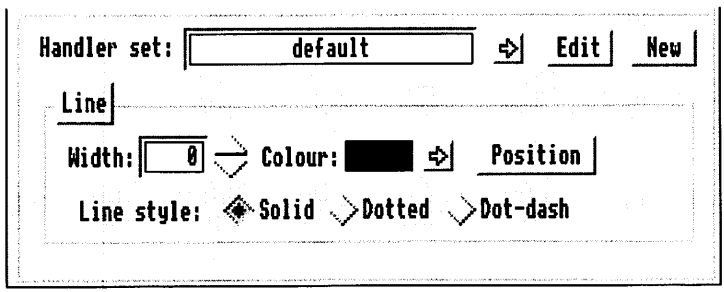
Rectangle elements require a handler set, outline and fill information. Selecting an outline will allow you to set the line width and colour, and corner style.

A line width of 0 signifies the line should be drawn using the thinnest width possible for the display device (for a high resolution printer this can be very thin).



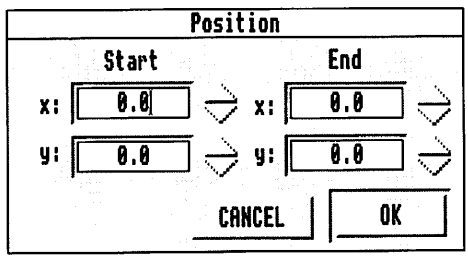
If the rectangle is filled you can choose a fill colour from the popup palette.

Select 'Line' from the type popup.



Line elements require a handler set, width, colour, style and position. The line width and colour are the same as described for rectangles.

Click on Position to open the line position dialogue.



This dialogue allows you to enter the start and end coordinates of the line. Lines are always fitted to these coordinates: you cannot specify a separate bounding box for the line.

Click 'Cancel' in the element dialogue to abandon the changes you have made to the `name` prompt element.

Now that you have a completed format it is time to use it. In the next chapter you will link each of the pieces together to produce a working address card.

10. Cards

A card is a window where the user can enter information to store in the database, and look at information previously stored. Cards use formats to define their layout and much of their operation.

S provides a number of commands related to cards. The full details of their use can be found in the S Reference. This chapter will show you how to implement a simple card and gives a basic look at handlers.

Controlling S-Base

S *code* can appear in programs, handlers, procedures and menus. S expressions can also appear in format elements to be evaluated and displayed. S gives you control over your application. Before you add any code to the address book application, you should understand how to control S-Base.

An application can be run by selecting 'Select–>Run' from the application list viewer. You can also click into any window of that application and press Shift-F9. S-Base will start by executing a program called `startup` in the program list viewer. If this doesn't exist your application will not do anything.

Once your application has started you must be able to stop it. Usually you will put a 'quit' option in your application and use that. Sometimes, however, your application can contain an error which causes it to go into an 'infinite loop' . The application appears to 'hang' and you cannot gain control to select the 'quit' option. In this case you can hold down both Ctrl keys at once. Before S-Base executes each command it checks for this and halts the application with an 'Escape' error.

When an error occurs, S-Base first displays the error then locates the resource containing it. If possible, S-Base will highlight the exact position of the error so you can quickly correct it. In this case, your application has stopped running but still has an *execution context* attached to it. This means that any windows opened by your application at the time of the error will remain open. You can remove an execution context from an application by clicking in any viewer for that application and pressing Ctrl-F9.

You do not have to exit an application or remove the execution context before running it again. Just press Shift-F9 whenever you like. S-Base will

remove the old context and start a new one, running the application.

Editing while S-Base is running

It is possible to edit an application while S-Base is running it. However this is useful only in rare circumstances. Changes you make will not always take immediate effect. You shold **never** delete resources while an application is running, since the application may require them. S-Base cannot detect resources dissappearing while it is running, and you may provoke a fatal error.

Comments

Double-click on the `startup` program in the program list viewer.

It will be empty except for a *comment* at the top. The vertical bar character '`|`' can be used to denote a comment line. Comment lines are ignored by S-Base and can contain any combination of text characters. The '`|`' must be the first character on the line (although it may be preceded by spaces) otherwise it will be taken as the 'bitwise or' operator.

Comments which appear on the same line as commands and functions, or which encompass a number of lines can be enclosed in `{ }`.

For example:

```
| This is a comment line

{ This comment can exist
  on a number of lines.}

print { these comments can appear almost anywhere }
"hello"
```

It is good practice to comment your programs and handlers so that they can be understood by other programmers and yourself at a later date.

Creating cards

Before creating a card it is important to ensure you have data to display in it. Enter the following lines of code into your `startup` program pressing return after every line:

```
recnum = 0
zero address
```

The variable `recnum` will be used later to store the record number of the

current `address` record in memory. The **zero** command is used to reset the current record of a file in memory to blank values (spaces and zeros). If you do not zero or load a new record S-Base will display the last record used. On start-up this will be blank anyway.

To create a card and display it on the screen S provides two commands **card new** and **card show**.

Enter the following text into the `startup` program:

```
card new ahandle,"address"
card show ahandle
```

The **card new** command creates a card using the format you have created, and stores the handle of the card in the variable `ahandle`. The handle of a card is the number S-Base uses to identify it. All commands that deal with cards require you to pass the handle as an argument.

The **card show** command opens the card and displays it on the screen. The only argument for the **card show** command is the handle of the card.

Once the card is open it is ready to be used. Append the following text to the bottom of the `startup` program:

```
card poll ahandle
```

When S-Base encounters the **card poll** command it will take control of your card and begin checking it for events generated by the user, such as mouse clicks and key presses. If an event occurs S-Base will call one of the handlers in the handler set attached to your card. If there is no handler set for the card S-Base will handle the event itself. S-Base will continue to poll your card until a handler tells it to stop.

After S-Base exits from the **card poll** it will return control back to your program. You will then need to remove the card. The **card dispose** command will hide the card and delete it. This will not affect the format or template used by the card.

Append the following text to the bottom of the `startup` program:

```
card dispose ahandle
```

This will be the last command to be executed in the `startup` program. When S-Base reaches the bottom of a program it returns to where the program was run from. In this case S-Base will return to the SDE and

terminate the execution of your application.

Press F3 to save your program and close the window. You may like to press Ctrl-F3 instead, which will save the whole application, including the program.

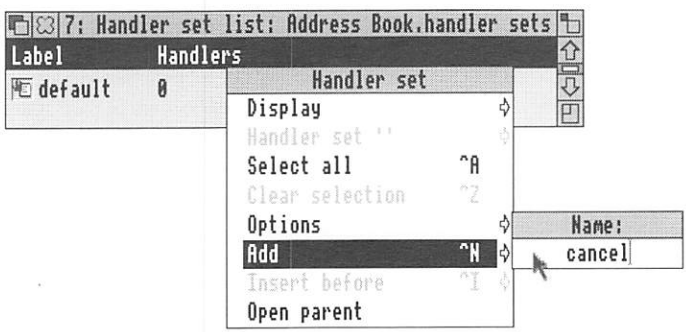
Saving from a text editor will not directly save the text to disc. Instead, it is treated like the 'OK' button in dialogues and only confirms the changes to the program.

Your program will now:

1. Create a card
2. Show the card
3. Use the card
4. Remove the card
5. Finish

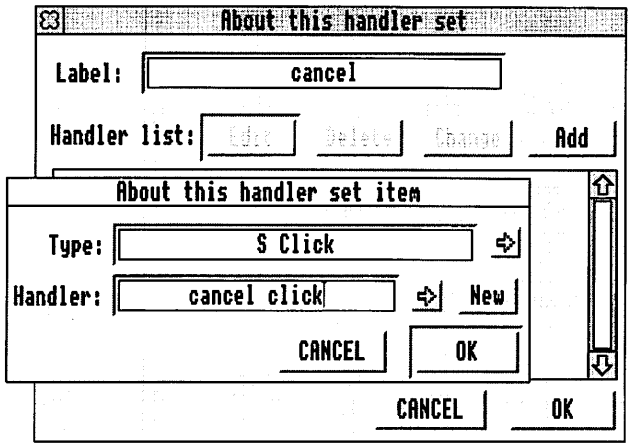
Creating handler sets & handlers

You will need to create a handler set to control your card. Open the handler set list and create a new set labelled `cancel`. This will be used to manage the `cancel` icon in the `address` card.



S-Base will create the handler set and open the handler set dialogue for you to edit it. This dialogue consists of a list of handlers and a number of control icons. At this stage the only active button is for adding handlers. Click on this and S-Base will open the handler set item dialogue. This dialogue is used for changing or attaching a handler to a handler set.

The type field defines the type of event to which this handler will be attached.



Events are generated by the system when the user does something, like clicking on an icon. Event types fall into one of two categories depending on how the event should be processed.

Raw event handlers receive the data relating to the event directly from the operating system in a raw form. The handler must interpret the data and take any necessary action to process the event. The use of raw event handlers requires a substantial knowledge of the operating system and should be restricted to operations S-Base does not support. For example, notification that the user has changed the printer driver can only be obtained through a raw event handler. Such events are seldom useful in an S-Base application.

S event handlers receive the data relating to the event from S-Base. This has been processed for the handler and appropriate S functions have been updated to help your application decide what to do. There is no requirement for your handler to interface with the operating system; S-Base will manage this aspect of the event. S event handlers should be used in all but a few situations.

A description of each handler type and related S functions will be given in the 'Handlers' chapter later in this tutorial.

Click on the type popup to view the menu of possible handler types. Select *S Click* from the menu. An S Click event is generated whenever the user clicks on a format element or format. Enter `cancel click` into the handler field and click on the 'New' button.

S-Base will create the new handler then attach it to the handler set, confirm the changes in the handler set and item dialogues, close the handler set dialogue, and display the handler in a text editor.

If a handler already exists it can be selected from the popup next to the handler field. You should then click 'OK' in both the handler set and item dialogues. The 'new' button is a short cut which does this for you.

Click into the text editor window and enter the following text into the handler:

```
card exit poll ahandle
```

The **card exit poll** command instructs S-Base to exit from the **card poll** without processing further events related to the card. The next time S-Base polls the card, it will see that it has been told to exit and the **card poll** command in your program will stop. S-Base will execute the next command. The result: clicking on the `cancel` icon in the `address` card will close the card and cancel any alterations to it. Save the handler with F3 and close it (Ctrl-F2 is the shortcut for closing editing windows).

Try creating another handler set called `save` with an S Click handler called `save click`. This handler set will be used for the `save` icon in the `address` card.

Enter the following text into the handler:

```
create address
card exit poll ahandle
```

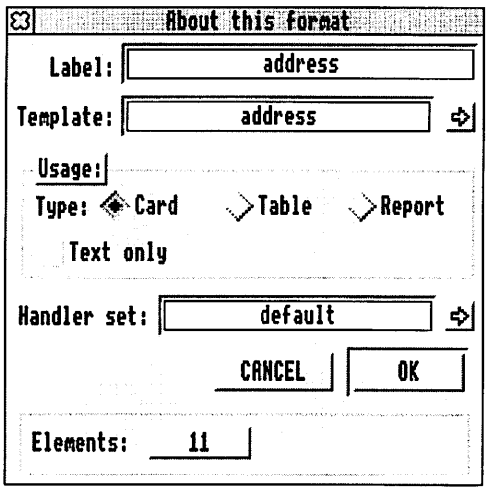
The **create** command is used to create a new record in a database file and must be followed by a file label. The result: clicking on the `save` icon in the `address` card will create a new record in the `address` file and exit the card (and the application).

Save the handler with F3.

Attaching handler sets

Try running your application with Shift-F9. The `address` card will open and appear to function correctly. However, clicking on the `save` and `cancel` icons will do nothing. You will not be able to exit from the card so click into an S-Base resource viewer and press Ctrl-F9. This will terminate your application and close any open files, etc.

You will need to attach the handler set to your format before S-Base can use the handlers you have just created. There are two ways of attaching a handler set to a format. The first way involves attaching the handler set to the entire format from the format info dialogue.

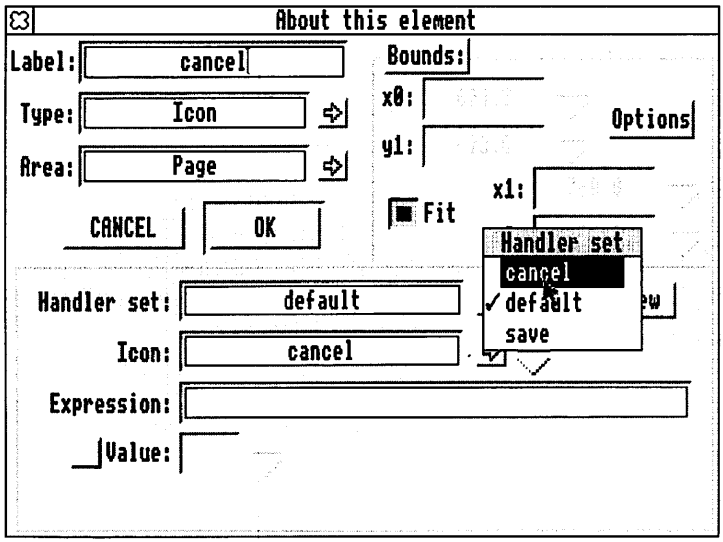


The format info dialogue displays the type of the format and two popups for attaching a template or handler set. AFG will have already set up the template and type. The handler set will be set to 'default'. This can be changed by clicking on the popup and selecting a handler set from the menu.

The format handler set will apply to every element in the format unless otherwise stated in the element. This method of attaching a handler set best applies to operations involving the entire format. For example, clicking on the Close box on the title bar relates to the whole card rather than any particular element.

For your `address` card the second method is recommended. This involves attaching the handler set to individual elements with the element info dialogue. This is the more flexible of the two approaches and allows you to set a different handler set for each element if desired.

Open the format layout editor for your format and double-click on the `cancel` icon. At the bottom of the dialogue in the icon information there will be a handler set field. Click on the popup and select `cancel`.



Click 'OK' to confirm your changes. Try setting the handler set for the save icon to `save`. Save your application using Ctrl-F3.

Run your application (Shift-F9) and test the `save` and `cancel` icons. Clicking the `cancel` icon will abandon the current data and quit the application while clicking `save` will create a new record then exit. Create a number of new records. You will use these in the next chapter.

The next step is to view the records you have created with the arrow icons. In the next chapter you will add, edit and view address records and print out a list of names.

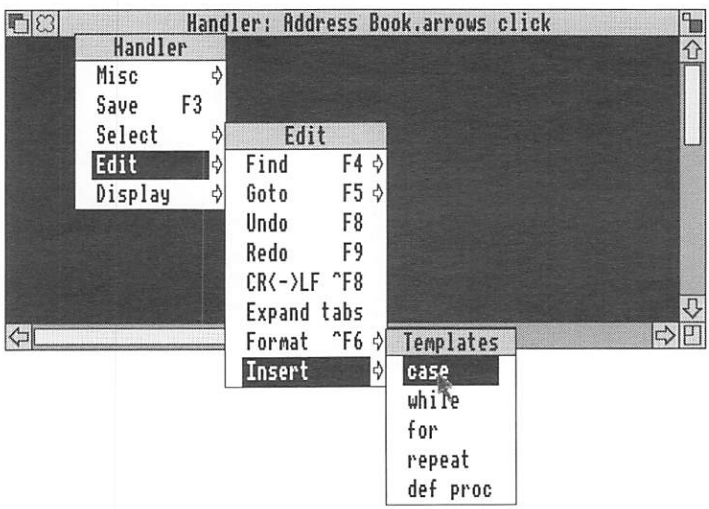
11. Control structures

A computer language requires a set of control structures to control the execution of its programs. Control stuctures are built from a number of S commands. This chapter will use show the correct use of these commands and further expand your address application.

Control structure templates

The text editor built into S-Base is very similar to packages such as Edit and SrcEdit, though it contains a few special features. One such feature is structure templates. Create a new handler set labelled `arrows` and an 'S Click' handler labelled `arrows click`.

Click into the window and open the text editor menu. Open the 'Edit–>Insert' submenu. This submenu displays a list of the *program templates* available in the editor. Click on 'case'.



S-Base will insert a blank **case** structure at the current cursor position. Text templates are useful for inserting commonly used commands or instructions.

For the advanced user, the text template file ('etemplates') can be found inside the S-Base directory. This file can be edited to add your own commonly used sections of code. A template must start with a label

prefixed by a dot. This label will be displayed on the templates menu. The template text should follow the label with a line feed at the end of each line. The sequence '\c' tells S-Base where to place the cursor after the template has been inserted. The template definition for the **case** structure is:

```
.case
case \c

when

otherwise

endcase
```

case..when..endcase

Modify the empty **case** structure in the `arrows click` handler to match the following code (you should indent the code after the 'when' lines as shown):

```
case @clickelementlabel
when "left arrow"     {left arrow clicked on}
   recnum -= 1    {move to the previous record}
   load address,recnum
   card update ahandle
when "right arrow"    {right arrow clicked on}
   recnum += 1    {move to the next record}
   load address,recnum
   card update ahandle
endcase
```

The **case** structure is the most commonly used structure for controlling the execution of code in a handler. The first part of the **case** structure is the **case** command. This command has an argument which is evaluated and compared with each of the following **when** commands in the **case** structure. The argument can be any valid S expression: a variable, a string, a number, etc.

The body of a **case** structure usually consists of a number of **when** commands. These are followed by a list of expressions which will be compared with the **case** argument. These must be of the same data type when evaluated. If any of these expressions match the **case** argument the code following that **when** and preceding the next will be executed, then control will step to the line after the **endcase**.

An optional **otherwise** command can be inserted below the last **when** command (before the **endcase**). This will be executed if none of the **when** commands match the **case** argument.

Case structures must be terminated by an **endcase** command.

The code above uses the **@clickelementlabel** function as the **case** argument. This function returns the label of the element clicked on by the user, as a string. It does not require the card handle to be passed as an argument (although it can be). S-Base will assume that you are referring to the card which called the handler if no handle is supplied.

A **when** command is included for each arrow icon. If the `left arrow` is clicked on, the value of **@clickelementlabel** will be `"left arrow"`. This will match the first **when** and its code will be executed. The current `address` record number will be decremented and a new `address` record loaded by the **load** command. Note that `x += 1` is short for `x = x + 1`. You can put most operators before an equals sign in this way.

The **load** command is used to load a new record into memory. Only one record for each file can be present in memory at a time, so the record which was previously in memory will be replaced. The **load** command has two arguments: the file label and the record number. A valid record number (from 0 to the number of records in the file) must be used or an 'Invalid record number' error will be generated. By convention, record 0 of a file holds the default record, generally blank, so loading this is the same as executing the **zero** command on the record.

Finally, the new record will be displayed in the card. The **card update** command will update the contents of the card and display it. **card update** can have a number of arguments. If the card handle is the only argument then every field and icon in the card will be updated. Individual elements can be updated by listing them after the card handle. For example,

```
card update ahandle,"name","home_address0"
```

will update only the `name` and `home_address0` fields in the `address` card.

Attach the `arrows` handler set to the `left arrow` and `right arrow` icons in the `address` format, and run your application.

if..then..endif

You can now scroll through the records you have previously created. If you scroll too far and attempt to load a record that does not exist S-Base will generate an error. Your application will need to check to see if a record exists before you attempt to load it.

Modify the `arrows click` handler to match the following code. You can edit it by double-clicking on one of the arrows with Ctrl and Shift held down.

```
case @clickelementlabel
when "left arrow"     {left arrow clicked on}
   if recnum > 0
      recnum -= 1    {move to the previous record}
      load address,recnum
      card update ahandle
   else
      vdu 7
   endif
when "right arrow"    {right arrow clicked on}
   if recnum < @recordcount address then recnum += 1
   load address,recnum
   card update ahandle
endcase
```

You will see that an **if..endif** structure has been added. There are two different forms to the **if..endif** structure. The first, used in the left arrow code is, known as a multi-line **if**. The second, used in the right arrow code, is a single line **if**.

Multi-line **if** structures consist of an **if** command, lines of code to be executed, and a terminating **endif**. The argument for the **if** is a logical expression. If the logical expression evaluates to true (non-zero) the code enclosed within the **if** structure will be executed.

An optional **else** command can be inserted before the **endif** and the code following this will be executed if the logical expression evaluates to false (0). In the example, if `recnum` is greater than 0, it will be decremented and the card updated. Otherwise the computer will beep (**vdu** 7).

Single line **if** structures consist of an **if** command, a **then** command, and some code. There are no **else** or **endif** components in this form.

The `right arrow` code uses the function **@recordcount**. This function returns the number of records in the file specified in its argument. If the

right arrow icon is clicked on the handler will check if the record number stored in recnum is less than the number of records for the file. If so it will increment recnum otherwise it will remain unchanged. The record referenced by recnum is then loaded and displayed. Note that this action is different from the left arrow, since clicking on the right arrow icon in the last record will destroy any changes made to it.

Save the handler and run your application and see if you can scroll beyond the limits of the address file.

The first record (0) of a file should always remain blank or contain default data. S-Base does not support use of this record other than for loading and saving it as the default record.

In the tutorial directory on your S-Base working discs you will find an application called '!Address1'. This is a modified version of your address book application. You will use this application to continue the tutorial. Remove your version of the address application (use the 'Select–>Remove' menu option in the application list viewer). If you have a hard disc, copy !Address1 onto it. Load !Address1 into S-Base.

In Address1 the startup program declares a new variable adding_rec. This will be set TRUE if the user is adding a new record and FALSE if he is updating it. The startup program will check to see if there are any records in the file and will set adding_rec according to the result. If there are no records in the file, the application will immediately go into 'add mode'.

There is an extra icon (add) in the address template and format. A new handler set add is defined and contains a handler add click. This is attached to the add icon. Add click zeros the current record, sets the adding_rec variable to TRUE, places the cursor in the name field, and updates the card. You should inspect this handler.

The **card set element** command places the cursor inside a field. There are two arguments to this command. The first argument is the card handle and the second argument is the label of the destination field.

The save handler has been modified to create or update a record depending on the state of adding_rec. A new command has been introduced in this handler. **update** is used to update the contents of an existing record on disc. **update** requires the file label and record number to be passed as arguments. The save icon no longer exits from the card when

it is clicked.

It is recommended that you study these changes carefully. It will be assumed that you have a good understanding of the operation and contents of this application.

for..next

The **for..next** control structure tells S-Base to execute a block of code several times. You will use this command to output every name in the `address` file to the command line.

From the format editor click on the paper options icon to open the template editor. Create a new icon in the `address` template labelled `debug` with an icon style `action` and text buffer override `Debug`. Return to the format editor and use the format icon creation tool to create a format icon with the same label and attach it to the template `debug` icon. Finally attach a handler set to it labelled `debug` and an S Click handler `debug click`. Enter the following code into your handler:

```
local x
for x = 1 to @recordcount address step 1
   load address,x
   print address.name
next

! load the old record back in
load address,recnum
card update ahandle
```

When the `debug` icon is clicked this handler will load each record from the `address` file and output it to the command line. It will finish by reloading the original address record and updating the card.

The **local** command is used to declare a list of temporary variables that only exist while the current object is active. When S-Base exits from the handler it will execute a '`clear x`' then restore the old value of `x` (if it existed before). All variables that are not used globally across your package should be made **local**.

The **for** command has been provided for situations where a block of code needs to be executed a given number of times. **For** consists of a variable assignment (`x = 1`) and a limit the variable can take (`to @recordcount address`). An optional **step** command which follows this can be added to specify the value to add to the variable each time around the loop. If this is

not given a default of 1 is used, so **step** could have been left out of the handler in this case.

The variable x is called the control variable and is initially assigned the value 1. S-Base then executes the block of code following it, until it finds a **next** command. S-Base will then loop back to the **for** command and increment x by 1. If x is less than or equal to the number of records in the address file, the loop will be executed again. If not, control will pass to the line after the **next** command.

S-Base also supports a variation on the **for** control structure. The **ffor** command is similar to **for** but it checks if the limit has been exceeded before it executes the first iteration (repetition) of the code. The above code would fail if there were no records in the file, since it would try to load record 1 and get an error.

Replace the **for** above with **ffor** to correct this. You can remove the step 1 as well. Attach the debug handler set to the debug icon and run your application. Clicking on the debug icon will open the command line and print a list of all the names. Add some records to make this work.

repeat..until

The **repeat..until** structure is also used to execute a block of code a number of iterations. Unlike the **for** structure the number of iterations is not fixed. The following code will produce the same results as the example above (you don't have to type this in):

```
local x
x = 1
repeat
   load address,x
   print address.name
   x += 1
until x > @recordcount address

| load the old record back in
load address,recnum
card update ahandle
```

The **repeat..until** structure consists of a **repeat** command followed by the code to be executed. The control structure is terminated with an **until** command. S-Base will repeatedly execute the code between the two commands until the condition after **until** is satisfied. S-Base uses the **repeat** command only as a marker to the start of the structure. The **until**

command has a logical expression as an argument. This is evaluated and the result used to decide the course of action. If the result is false (0) S-Base will loop back to the **repeat** command and execute another iteration. If it is true (non-zero) it will exit from the **repeat** control structure and execute the next line. It is possible for S-Base to get into an *infinite loop* if the result is always false (0).

In the example above x is used to keep the record count. It is compared with `@recordcount address` to decide whether all the records have been printed. x is incremented after each iteration. This code does not work correctly if there are no records in the file.

while..endwhile

The **while..endwhile** structure is similar to the **repeat** structure. The structure consists of a **while** command followed by the code to be executed. The control structure is terminated with an **endwhile** command.

You can see another variation in the way the debugging routine can be implemented.

```
local x
x = 1
while x <= @recordcount address
   load address,x
   print address.name
   x += 1
endwhile

| load the old record back in
load address,recnum
card update ahandle
```

The **while** structure is effectively a **repeat** structure in reverse. The logical test following the **while** is performed before the iteration is executed. While the expression evaluates to true (non-zero) the block of code will be executed. If the test fails S-Base will exit from the structure and execute the next instruction after **endwhile**.

In the example x contains the record number to be loaded and is first compared to see if it is a valid record. If so it will load and print the name in the command line. This code works correctly if there are no records in the file.

def proc..enddef

The **def proc..enddef** structure is used to define a *procedure* in your program, procedure or handler resource. This command will only define the procedure and will not execute any code. The structure consists of a **def proc** command followed by the procedure code. The structure is terminated by an **enddef** command.

The **def proc** command defines a unique procedure label. This label is treated by S-Base as a variable of type procedure. Procedures not resident in the procedure resource should first be declared as a *local variable* in the program/handler in which they reside. S-Base will not allow you to define a procedure twice or two procedures with the same name. Declaring a procedure local will cause the procedure variable to be destroyed when you leave the program/handler.

The procedure definition can contain a number of parameters surrounded by brackets. These variables are passed into the procedure and automatically declared local to the procedure. For example,

```
local hello

def proc hello(message, lucky_num)
   print "The message is ",message
   print "My lucky number is ",lucky_num
enddef
```

Procedures are called with the **proc** command. The **proc** command has a procedure label as its only argument. The **proc** command must contain an identical number of parameters as the procedure definition. The example above would be called by:

```
proc hello("Hello there", 7)
```

A procedure can also return a value like an S function. The **return** command will return its argument to where it was called. This can be ignored or stored and used. The **return** command will return immediately to where it was called. If a **return** command is placed in the middle of a procedure, the procedure will exit upon executing the command. For this reason **return** commands are usually the last line of a procedure. For example, the following procedure adds one to its argument and returns it:

```
local add_one

def proc add_one(number)
```

```
   number += 1
   return number
enddef

a = proc add_one(b)
```

The **return** command can be used to return a number of values. Adding **return** to the front of a procedure parameter will return the value of the parameter when the procedure is exited. This is passed back to the parameter used in the **proc** command. The parameter in the **proc** command must be a variable. For example,

```
local add_many

def proc add_many(return add1, return add2, add3)
   add1 += 1
   add2 += 2
   add3 += 3
enddef

value1 = 0
value2 = 0
value3 = 0
proc add_many(value1, value2, value3)
```

After executing this code `value1` will equal `1`, `value2` will equal `2`, `value3` will contain `0`. The new values of `value1` and `value2` were returned from the procedure through the parameters. The variable `value3` was passed to the procedure as a normal parameter and was automatically declared local. Hence, it retained its original value after exiting the procedure.

S-Base allows an entire array to be passed as a procedure parameter. However, the array must be passed as a **return** parameter. Arrays can consume large amounts of memory and it is impractical for S-Base to make a copy of the array when it is localised. This restriction does not apply to individual array elements.

Which control structure?

Each control structure is designed to be used in a different situation. **if..then** and **case..when..endcase** execute code conditionally and are only used for single iterations of code, whereas **for..next**, **ffor..next**, **repeat..until**, and **while..endwhile** can perform multiple iterations.

When the only possible outcomes of a test are true and false **if..then** should

be used. A **case..endcase** could also be used in this situation but this is unnecessary.

When there are a number of possible outcomes from a test a **case** structure should be used. A nested **if..endif** can be used to simulate a **case..endcase** structure though this quickly becomes very messy.

The **for..next** and **ffor..next** control structures are designed for doing a fixed number of iterations and are the easiest and cleanest of the structures to use for this purpose. They should be used where you will not be required to prematurely exit from the control structure. **ffor..next** provides protection against executing the loop once when it should not be executed at all.

The **repeat** and **while** structures are not fixed to a control variable. These structures rely on a logical test to determine when to exit, and give far more flexibility. The choice of which to use depends upon when the logical test should be made. Where a condition must be fulfilled before the code block is executed, **while..endwhile** should be used. Where a condition should be checked after executing the code block, **repeat..until** should be used.

The **def proc..enddef** structure is designed to call code that will be used a number of times in an application . For example,

```
a = 1
b = 10
c = a + b^2
a = @cos(c / 16 + 4)
print "a = ",a
b = 18
c = a + b^2
a = @cos(c / 16 + 4)
print "a = ",a
b = -7
c = a + b^2
a = @cos(c / 16 + 4)
print "a = ",a
```

could be replaced by:

```
def proc calculate(a,b)
   local c
   c = a + b^2
   a = @cos(c / 16 + 4)
   print "a = ",a
   return a
```

```
enddef

a = 1
a = proc calculate(a,10)
a = proc calculate(a,18)
a = proc calculate(a,-7)
```

Grouping sections of code into procedures will reduce the size of an application and make for 'cleaner' code.

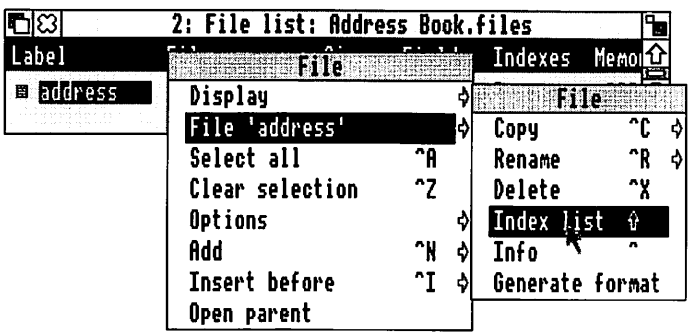
12. Indexes

Indexes are an important part of any good database. They are used to search for data and define the order in which it is retrieved. An index is a list of record numbers arranged in a defined order. Instead of searching every record in a large database it is possible to look up the index and find a record very quickly. There is no limit to the number of indexes you can attach to a file, although there is a speed penalty for each index when adding or updating records.

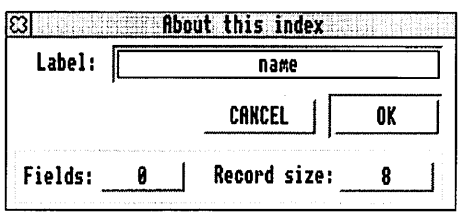
In this chapter you will add an index to the `address` file and change the arrow icons to work in alphabetical order. S index functions will be discussed and examples given of how they should be used.

Adding indexes

To add an index you will need to open the file list viewer. Click Menu over the `address` file and select 'Index list' from the file submenu. You can also open the Index list viewer by double-clicking on `address` with Shift held down.

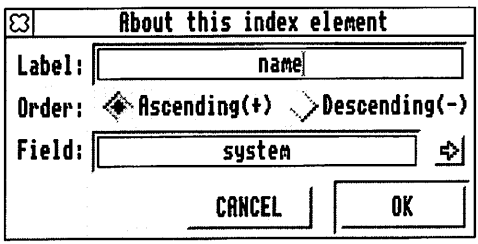


Add an index labelled `name` to the index list in the usual way. S-Base will create the index and open the index info dialogue.



This dialogue only allows you to alter the label of the index. It displays the number of fields and the record size of the index.

Double-click on the `name` index to open it. Add an index field labelled `name` to the index in the usual manner. S-Base will add the field and open the index element dialogue.



This dialogue contains order icons and a field popup. Indexes can be ordered in ascending (1, 2, 3...) or descending (99, 98, 97...) order.

Click on the field popup, select the `name` field and click 'OK'. You have now created an index labelled `name` which orders the records in ascending alphabetical order based on the `name` file field.

Indexing on more than one field

An index can contain any number of fields as long as the record size is less than 1000 bytes. The larger the record size grows the less efficient the index becomes. Indexes can only contain simple single-value fields: arrays and structures are not allowed.

Indexing on more than one field allows you to search on more than one criterion at once. It can also allow you to order the records on a number of criteria. For example, the `customer` file in Video is indexed on the `surname` and `initial` file fields. You can search the `customer` file with a particular surname and initials instead of finding just the first customer with that surname. The `customer list` in Video uses this index to print an ordered list of customer names and addresses. Each record would be ordered by surname first and all identical surnames would be ordered by initials. For example:

```
Simes       JL
Smith       AJ
Smith       BM
Smith       CP
Thames      K
```

If you only indexed on surname, then the three Smiths might appear in any order. In fact, S-Base puts them in the same order as they appear in the file, lowest record number first. In descending order this is reversed.

Traversing an index

S provides two functions for traversing up and down an index. Alter the `arrows click` handler to match the following:

```
local new_recnum

case @clickelementlabel
when "left arrow"     {left arrow clicked on}
   new_recnum = @previous (address,"name",recnum)
   if new_recnum <> 0  {move if valid}
      recnum = new_recnum
      load address,recnum
      card update ahandle
   endif
when "right arrow"    {right arrow clicked on}
   new_recnum = @next(address,"name",recnum)
   if new_recnum <> 0  {move if valid}
      recnum = new_recnum
      load address,recnum
      card update ahandle
   endif
endcase
```

A temporary variable `new_recnum` has been added to store the number of the record you are traversing to. Clicking on the `left arrow` icon finds the previous record number in the index and stores it in `new_recnum`. **@previous** has three arguments: the file label/number, the index label/number, and the record number to start the search from. These must be surrounded in brackets. If the record number given is the first record in the index **@previous** will return 0 since there is no previous record. The code above updates `recnum` only if **@previous** returns a valid record number.

Clicking on the `right arrow` icon finds the next record number in the index and stores it in `new_recnum`. The opposite function, **@next**, has the same arguments as **@previous**. If the record number is the last record in the index **@next** will return 0. The code checks to see if the user has reached the end of the index and does not allow them to advance any further.

Index and file numbering

When you want to specify a file, you generally just write its label without quotes. In fact, S-Base recognised this as a structure. You can also write the label with quotes, e.g. `@next ("address", "name", recnum)`.

Indexes, as with almost all resources, can be written either as a label in quotes or as an integer. Usually the label is more convenient since the index number depends only on the physical order of the index in the index list viewer. Since you know that the `name` index is the first (#0) in the index list viewer, you could write:

```
@next(address,0,recnum)
```

S-Base understands this as the same thing. If you want to use numbers for indexes you should use constants. You could define a constant called `NAME_INDEX` and write:

```
@next(address,NAME_INDEX,recnum)
```

As you can see, the label is much more convenient.

The S Reference refers to a *filespec* as a place where you must specify a file and an *indexspec* as a place where you must specify an index.

Finding records

The purpose of a database is to store and retrieve information. S-Base is very good at retrieving information. S provides a number of commands and functions to this end. Of these, **@find** and **@findn** are the most commonly used. Both of these functions find a record in a file using a given index and return its record number. **@find** searches for an exact match and will return 0 if not found. **@findn** finds the nearest matching record. It will always return a valid record unless there are no records in the file (in which case it returns 0).

Both functions have the same arguments and these must be surrounded in brackets. The first argument is always the filespec. This is followed by the indexspec. The remainder of the arguments give the values to search for. The number of arguments depends on the number of fields in the index. For example, the `customer` file in Video has an index of two fields and thus a maximum of two pieces of data can be provided. For example,

```
x = @find(customer,0,"smith")
```

This will only search on the `surname` field in the index. S-Base will always find the first matching record in index order. This will be Smith AJ, using the data given in the example above.

```
x = @find(customer,0,"smith","BM")
```

This will search on the `surname` and `initial` fields in the index. The first match in this case will be Smith BM.

A number of rules apply to string searches:

1. For strings, searching is not case sensitive.

2. Trailing spaces in a string argument are considered in a search. For example, searching for `"smith  "` will not find `"Smithy"`.

3. S-Base will compare equal numbers of characters in the argument and the file field being searched. For example,

```
x = @find(customer,0,"s")
```

will only compare the first character of the `surname` file field. 'Simes' will match this.

In the next chapter you will implement a simple search function using **@findn** and explore some of the more complex areas of cards.

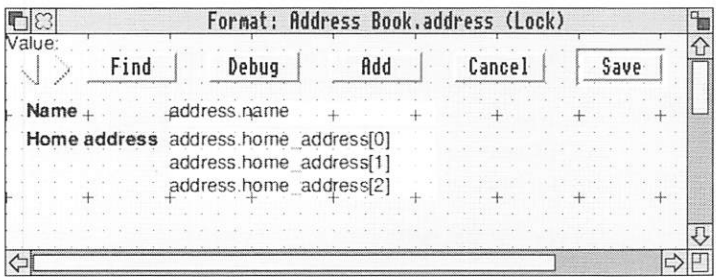
13. More on cards

So far in this tutorial you have opened an `address` card and allowed the user to create, edit and browse through records. In this chapter you will begin to explore some of the more interesting areas of cards. You will open a second card and learn how to use a keypress handler. After completing this chapter you will be able to use most aspects of cards.

Scared cards

The key to a successful database application is being able to quickly and easily find any piece of data. The previous chapter demonstrated the power of **@find** to quickly search for a data record. Now you can add a search facility to your address application using that function.

Add an icon labelled `find` to the `address` template and format.



You will need to resize the template to make the icons fit. Create a format icon called `find` in the format editor to match the `find` icon.

Create a new handler set labelled `find` and an S Click handler `find click`. Enter the following text and attach the handler set to the `find` icon:

```
local new_recnum=-1

zero address   {initialise the name field}
card new fhandle,"find record"
card show fhandle
card poll fhandle,close "cancel"
card dispose fhandle
```

This code will initialise and open a *scared card* to search for an `address` record. `new_recnum` will store the record number found.

Scared cards behave as normal cards except they are frightened easily! In some situations it is desirable to temporarily open a card and close it if the user attempts to click into another card. RISC OS menus are an example of a scared object.

S-Base supports scared cards by providing a **close** command after the **card poll** command. The **close** command tells S-Base to simulate an icon click if the user clicks into another card. In the example above the icon to be 'clicked' is `cancel`. S-Base will call the `find record click` handler attached to the `find record` card with **@clickelementlabel** set to `cancel`. The click handler will process the event as if the user had first clicked on the `cancel` icon in the `find record` card. S-Base will not automatically close the scared card. The `find record click` handler will need to do this in the usual manner (if it is desired). S-Base will then process the original click event and call the appropriate handler.

Nested cards

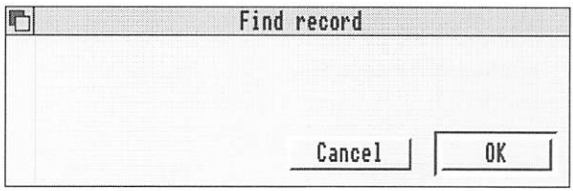
It is important to note the action S-Base takes when encountering the second **card poll** command. S-Base will start polling the find card, and will do so until it receives a **card exit poll** command for this card. S-Base will still operate the `address` card so, for example, a **card update** on it will still work.

The result of this is that S-Base must exit the `find record` card's **card poll** before it will take action on any **card exit poll** for the `address` card. **card poll** commands must be exited newest poll to oldest poll.

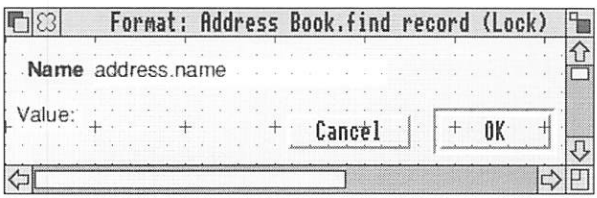
After S-Base has exited from the `address` card it will return to the `find click` handler and execute the remainder of the handler. S-Base then resumes polling the `address` card. S-Base allows any number of **card poll** commands to be nested in this fashion.

Create a new template labelled `find record` using the browser window style. Click on 'Title' to tick (override) it and enter `Find record` as the buffer. Make sure 'Text' is clicked too (just above 'Buffer').

Click 'OK', open the template and use the 'Try out' tool to resize it smaller.



Add `OK` and `Cancel` icons labelled as shown. Use the icon styles provided for each. Create a format called `find record` containing `address.name`, by selecting just the `name` field from the `address` file and using AFG. You will also need prompts, icons and a value bar. Use `find record` as the template.



Change the options for the `name` field to switch off automatic recalc. Your keypress handler will handle this recalculation.

Create a new handler set labelled `find record` and attach it to both the `ok` and `cancel` icons. Add an S Click handler labelled `find record click` to the handler set. Enter the following text into the handler:

```
case @clickelementlabel
when "cancel"
   load address,recnum   {restore the old record}
   card update ahandle
   card exit poll fhandle {exit find record}
when "ok"
   if new_recnum <> -1 then recnum = new_recnum
   load address,recnum  {load the new record}
   card update ahandle
   card exit poll fhandle {exit find record}
endcase
```

If the user clicks on the `cancel` icon the handler will reload the original `address` record, update the `address` card and exit from the **card poll**.

If the user clicks on the `ok` icon the handler will make the found record the current record, load and display it, and exit from the **card poll**.

Having successfully completed these additions you should be feeling quite confident with S-Base. If you have had difficulties you can compare your results with the !Address2 application. This can be found in the 'tutorial' directory on your S-Base working discs.

Keypress handlers

Add an S Keypress handler `find record press` to the `find record` handler set.

An S Keypress event is generated whenever the user presses a key inside a card. S-Base calls the keypress handler and sets up a number of special functions to help you handle the event.

Enter the following code into your keypress handler:

```
if @justaltered
   new_recnum = @findn(address,0,@trim @elementvalue
(fhandle,"name"))
   load address,new_recnum
   card update valuebar fhandle,"Match is "+address.name
endif
```

The first new function you will encounter in this code is **@justaltered**. This function returns true (1) if the key just pressed has altered the value of the format element. The handler checks to see if the `name` field has been altered and if so searches for a new match. It would be inefficient to search for a new record when the user has just pressed a cursor key.

@elementvalue returns the current value of the element's data entry buffer. It requires the card handle and element label to be given as arguments. It is important to read the search data from the data entry buffer because S-Base does not update `address.name` with the new key until after it has called the keypress handler (in fact since manual recalc is selected, it won't update the expression at all).

The function preceding **@elementvalue** is **@trim**. This removes all trailing spaces from a string. This will need to be done since **@findn** is sensitive to trailing spaces.

After finding a match the handler will load the new `address` record into memory. It would now be desirable to display the match that has been found. **card update valuebar** is used to display a value in the value bar. In this example it displays `"Match is "` followed by the `name` field in the `address` record.

You may remember that you switched off the automatic recalc option for the `name` field. When automatic recalc is on S-Base will automatically display the current value of the data entry buffer in the value bar. When it is off S-Base will rely on the application to set up the value bar using **card update valuebar**. Also, automatic recalc must be off otherwise the data entry buffer will automatically write into `address.name` after exiting the handler. This will interfere with the value of `address.name` being displayed in the value bar.

To finish off your search function attach the `find record` handler set to the `name` field. Run your application and try finding a name.

Confirmed exit

Your application will now contain a reasonable amount of functionality. It is important to check that some of this functionality does not accidentally destroy data. Try editing a record and scrolling to another record without saving the changes. The changes will be lost and you will have to enter them again.

Insert the following line at the beginning of your `startup` program:

```
card_altered = FALSE
```

This line initialises a variable in which you will store the state of the card. This variable will be set `TRUE` if the card has been altered.

Create a new handler set labelled `address fields`. Attach a keypress handler labelled `address press` to the handler set. Enter the following line of code:

```
if @justaltered then card_altered = TRUE
```

If the field has just been altered the keypress handler will set this field. Attach the `address fields` handler set to each field in the card.

S-Base provides a function to check if a card has been altered. The **@altered** function returns `TRUE` if any field in a card has been altered and `FALSE` if not. This function should only be used if the card will be disposed of after the record has been saved. If the card has been altered **@altered** will remain true (1) until the card has been destroyed.

Before moving to another record you should prompt the user if the record has been altered. The user can decide whether they would like to abandon

the changes.

Open the procedures list viewer and create a new procedure resource called `address`. Type the following procedure definition into this resource:

```
def proc ok_to_continue
   local reply=0
   if card_altered
      error 1,"Abandon data alterations?", 3, reply,
"Warning from Address Book"
      if reply = 1
         card_altered = FALSE  {reset the state of the
card}
         return TRUE
      else
         return FALSE
      endif
   else
      return TRUE
   endif
enddef
```

This procedure checks the state of the card. It will display an error message prompting the user for action if the card was altered. `TRUE` will be returned if 'OK' was clicked and `FALSE` if 'Cancel' was clicked. `TRUE` will also be returned if the card has not been altered.

The **error** command can have up to five arguments. The first two arguments are the error number and the error message. These are both mandatory and you can decide these for yourself. The remaining three arguments are optional. These are the flags, reply variable and title. The flags determine whether 'OK' and 'Cancel' icon appear in the error dialogue. The reply variable contains the user's response: 0 = no response, 1 = OK, 2 = Cancel. The title is the text displayed in the title bar of the error dialogue.

Change the `cancel click` handler to match the following:

```
if proc ok_to_continue then card exit poll ahandle
```

The handler will exit the poll if it is OK to continue with the Cancel operation.

Change the `find click` handler to match the following:

```
local new_recnum=-1

if proc ok_to_continue
```

```
   zero address   {initialise the name field}
   card new fhandle,"find record" {create the find card}
   card show fhandle
   card poll fhandle
   card dispose fhandle
endif
```

The `find record` card will only be opened if it is OK to continue with the Find operation.

The `left arrow` and `right arrow` components of the `arrows click` handler should look like this:

```
when "left arrow"   {left arrow clicked on}
   if proc ok_to_continue
      new_recnum = @previous(address,0,recnum)
      if new_recnum <> 0
         recnum = new_recnum
         load address,recnum
         card update ahandle
      endif
   endif
when "right arrow"   {right arrow clicked on}
   if proc ok_to_continue
      new_recnum = @next(address,0,recnum)
      if new_recnum <> 0
         recnum = new_recnum
         load address,recnum
         card update ahandle
      endif
   endif
```

Alter your handler to match the code above. Insert the following line after the **endif** in the `save click` handler:

```
card_altered = FALSE
```

This line will mark the card as unaltered when the user clicks on 'Save'. Run your application and check that it works correctly.

Protected icons

The checking you have just added is fine for editing an existing record though it is not practical in all situations. What should you do if the user clicks on the `right arrow` icon while adding a new record? What is the next record of a record that does not exist yet? The best method of getting round these problems is to prevent the user from creating them. S-Base has two commands to control the use of icons and fields.

Append the following line to the end of the `add click` handler:

```
card set protect ahandle, "left arrow", "right arrow",
"debug", "add", "find"
```

The **card set protect** command changes the state of an icon or field so that it cannot be selected (it is greyed out). The example above disables all the icons in the `address` card except `save` and `cancel` while you are adding a record.

Insert the following line between the **card show** and **card poll** commands of your `startup` program:

```
if adding_rec then card set protect ahandle, "debug",
"add", "find"
```

This line will set up the card for adding records if there are no records in the database when you start up your application.

Append the following line to the end of the `save click` handler.

```
card clear protect ahandle,"left arrow","right
arrow","debug","add","find"
```

The **card clear protect** command changes the state of an icon or field so that it can be selected. This is the reverse of **card set protect**. The example above enables the protected icons after you have finished adding a record. The **card clear protect** command will not affect the state of elements that have protect set in their options.

It is good practice to use protection to prevent the user from performing an operation that is certain to fail or produce an error. The `left arrow` and `right arrow` icons are good examples of this. If the user cannot scroll in a direction any further, the arrow icon should be protected. The user will see that he cannot scroll any further and not attempt to.

Open the procedure library labelled `address` and enter the following procedure definition at the start:

```
def proc setup_arrows
   if @previous(address,0,recnum) <> 0
      card clear protect ahandle,"left arrow"
   else
      card set protect ahandle,"left arrow"
   endif
   if @next(address,0,recnum) <> 0
      card clear protect ahandle,"right arrow"
```

```
   else
      card set protect ahandle,"right arrow"
   endif
enddef
```

This procedure checks to see if there is a previous or next record and sets the state of the `left arrow` and `right arrow` icons accordingly.

Insert the following line before the **card poll** command in the `startup` program:

```
proc setup_arrows
```

Before S-Base begins polling the `address` card this code will set up the `left arrow` and `right arrow` icons to their correct state. If the user cannot scroll in a direction then that arrow icon will be protected. Otherwise the arrow icon will be available.

Because all the invalid arrow icons in the card have been deselected it is unnecessary to check a browse operation in the `arrows` handler. Instead your code should check the validity of the next move and set the arrow icons accordingly.

Append the following line to the `find record click` and `save click` handlers:

```
proc setup_arrows
```

These handlers alter the current record in memory thus requiring you to set up the arrow icons for the new record. You may wish to remove the `"left arrow"` and `"right arrow"` arguments from the **card clear protect** command in the `save click` handler and the **card set protect** command in the `startup` program. The `setup_arrows` procedure will set their state.

The `left arrow` and `right arrow` components of the `arrow click` handler should look like this:

```
when "left arrow"     {left arrow clicked on}
   recnum = @previous (address,0,recnum)
   load address,recnum
   card update ahandle
   proc setup_arrows
when "right arrow"
   recnum = @next (address,0,recnum)
   load address,recnum
   card update ahandle
```

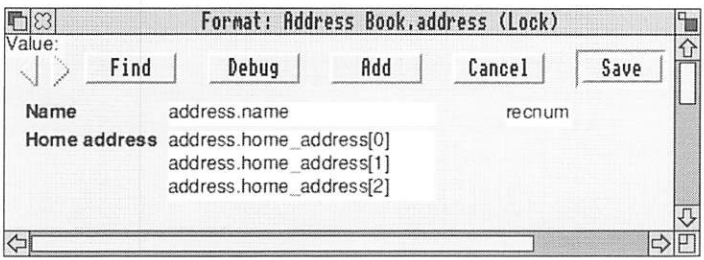
```
proc setup_arrows
```

Run your application and experiment with the `add`, `left arrow`, and `right arrow` icons.

Protected fields

The next addition to your `address` card will be a field displaying the record number of the current `address` record.

Create a new element in the address format labelled `recnum`. It should be a format field as format prompts are fixed and cannot be updated when the record number changes. Use the font style `field`.



The font style actually used will be `field.protect` because this is a protected field. When a field is protected S-Base will attempt to use a protected version of the font style. Protected font styles contain the font style label followed by '`.protect`'. If S-Base cannot find a protected version it will use the normal font style. You can easily define how a field will look when it is protected.

Enter `@record address:-5` as the expression for the field.

The function **@record** returns the record number of the current record in memory for the specified file. This will change whenever a record is loaded or created. The '`:-5`' is formatting information and will be explained later in this tutorial.

The 'automatic recalc' option should be off so that S-Base does not try to write the contents of the data entry buffer into **@record** and fail. The 'protected' option should be on to prevent the user entering the field. There will be no handler set attached to the field.

Append the following line to the end of the `save click` handler:

```
card update ahandle,"recnum"
```

You will need to update the `recnum` field every time the value of **@record** changes. The value of **@record** will change after a new record has been created. The `arrows click` and `find record click` handlers update the entire card already.

The `address` card is now complete and ready for use. In the next chapter you will add a main menu and expand your application.

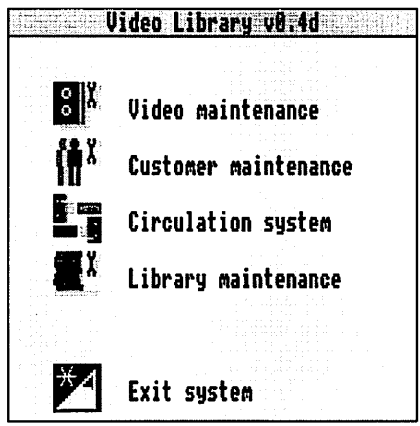
14. Program menus

The next step is to expand your application from a single card to include other facilities. To give the user a choice of options you will need to create a menu to choose from. This chapter will cover program menus.

Examples in this chapter will build on !Address3 found in the tutorial directory on your S-Base working discs. Replace your old address application with this.

Creating a program menu

Program menus are a 'no maintenance' version of cards. They require no handlers, formats or templates and are completely automatic. They are created, shown and disposed of in the same fashion as cards.



Load the `startup` program and examine the structure of it. The `address` card code has been grouped into the procedure `address_card`. All variables used by the `address` card have been declared local to the procedure.

Local variables are essential in applications that contain many parts. If a variable is not declared local to a procedure it could be accidentally changed by another procedure using a variable with the same name. This could produce unusual results or errors in your application, known as side effects.

The program menu must first be created and items added to it. It should be

shown, polled and then usually hidden while an option is being executed. The program menu should be disposed when you have finished with it. All of this can be performed with the **pmenu** commands.

pmenu new creates a new program menu. After creating the program menu S-Base stores the menu handle in the first argument of the command. The first argument of all **pmenu** commands must be the menu handle. The second argument of **pmenu new** is the title bar text. This is followed by the number of columns you would like the program menu displayed in. The last argument is the flags. See the S Reference for a list of these.

pmenu insert inserts items into the program menu. It requires the name of an icon, a key, some text and optional flags. Some of these arguments will be ignored depending on the flags in **pmenu new**. Consult the S Reference for a full description of this command.

pmenu show will show the program menu. The menu handle is the only argument.

pmenu poll will poll the program menu. Program menus will exit from the poll once a menu item is selected. The number of the item is returned in the second argument. Menu items are numbered from zero onwards and are not related to the keys provided with **pmenu insert**.

pmenu hide allows you to hide a program menu. The menu is not deleted or lost; it is just removed from view. **pmenu show** will put it back on the screen.

pmenu dispose will hide the program menu and delete it.

The `startup` program creates and displays the program menu until an option is selected. The program menu is then hidden and the option executed with a **case...endcase** control structure. If the last option is selected the variable `quit` is set to `TRUE` and execution exits the **repeat..until** structure. The program menu is disposed of and the application terminates. The `address list` option on the menu has not been implemented yet and an appropriate message is produced.

Try running your application. You will see it is shaping into a fairly professional-looking package.

In the next chapter you will create a report and implement the `address list` option on the program menu.

15. Reports

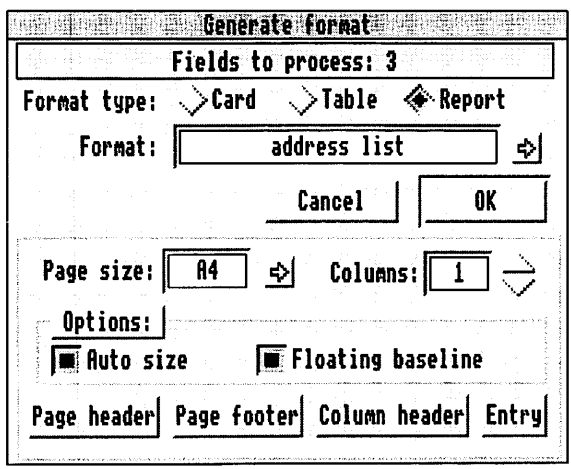
In the previous chapters you have explored the various areas of a database application. Initially there is the creation and editing of data. Next there is a need to search for data and display it. The final area of database operations is the processing of data its output it in a structured form. This is usually sent to the screen or a printer using a *report*. In this chapter you will create a basic report.

Creating reports

Reports are controlled by a format with similar components to cards. As reports are for display only and do not allow input from the user there is no need for icons and templates. S-Base describes the dimensions of a report with simple page size information attached to the format.

The first step in producing a report is to create an empty handler set labelled `address list`. You will use this handler set in AFG. You will find this technique can save a lot of time when creating cards with AFG. AFG will automatically attach the handler set to each of the elements in the format which need it. Select the `address` file and start AFG.

Select the report option at the top of the AFG dialogue. The bottom half of the dialogue will adjust to display report generation icons. You will notice the absence of the template field.



Enter `address list` as the format name.

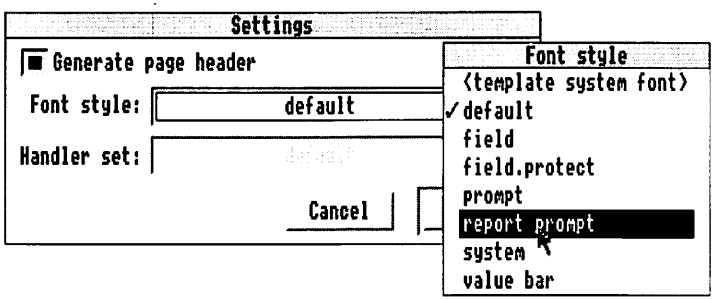
The next field is for the page size. Clicking on the popup icon will display a menu of the standard page sizes. A4 is selected by default and is suitable for this report. The page size can be altered later in the format layout editor.

There is a field for the number of columns on the page. This allows you to create a specified number of equal-width columns. These columns may be resized later in the format layout editor. The output in the report will flow down the first column before flowing to the next. Column headers will be printed at the top of each column if this option is switched on.

There are two options for report generation. 'Auto size' is the same as the option used in card generation. 'Floating baseline' sets the floating baseline option in the element option dialogue. This allows an element's baseline to extend down the page if there is insufficient room to print its contents.

There are a number of setting icons at the bottom of the dialogue. These perform a similar function to those found in card generation.

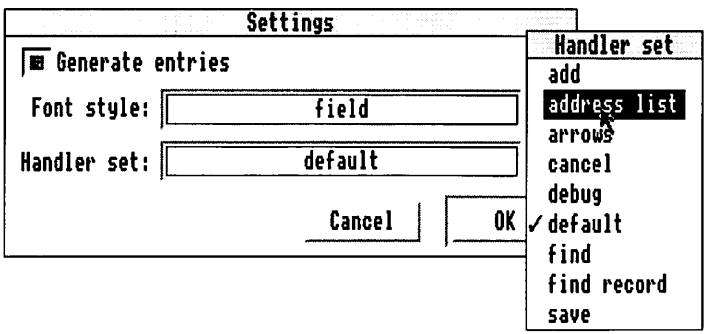
Selecting 'page header' will create a default header at the top of the page. The header consists of a header area and a prompt. The prompt must be enclosed by the area. Set this option and use `report prompt` as the font style.



Selecting 'page footer' will create a default footer at the bottom of the page. The footer consists of a footer area and a prompt. The prompt must be enclosed by the area. Set this option and use `report prompt` as the font style.

Selecting 'column header' will create a header at the top of the first

column. The column header consists of a column header area containing a number of prompts describing the file fields. The prompts must be enclosed by the area. S-Base only requires one column header to be defined and this must be positioned above the first column. S-Base will duplicate the header over the remaining columns when the report is executed. Set this option and use `report prompt` as the font style.



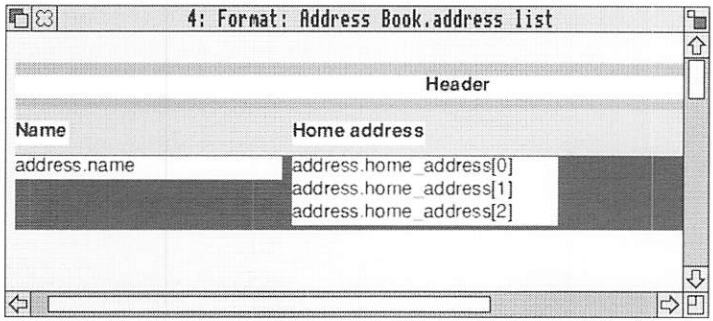
Selecting 'entry' will create a *report entry* in the first column. The entry consists of an entry area containing fields. The fields must be enclosed by the area. An entry is the data to be printed out in the report. S-Base only requires one entry to be defined and this must be positioned inside the first column. S-Base will duplicate the entry down the columns when the report is executed. Set this option and use `address list` as the handler set. Use `field` as the font style.

The area elements in your format describe the amount of space taken up on the page by each component of the report. They also describe to S-Base where they should be positioned in relation to each other. It is important to ensure that any element associated with an area falls within the bounding box of that area. Area elements are not printed when the report is executed.

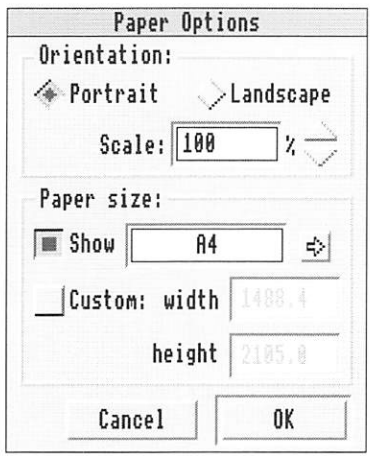
Click 'OK' and generate the format. S-Base will create the format and open the format layout editor.

Report layout

You will notice the layout of a report is completely different from that of a card. Before you examine this you will need to examine the changes to the format layout editor.



The selector and icon tools are not used in reports and have been deselected. Click on the paper options tool.



The paper options dialogue allows you to set information relating to the paper dimensions and orientation. Reports can be printed out in portrait format (down the page) or landscape format (across the page). The report can be blown up or shrunk down using the scaling option. This will not affect the paper size.

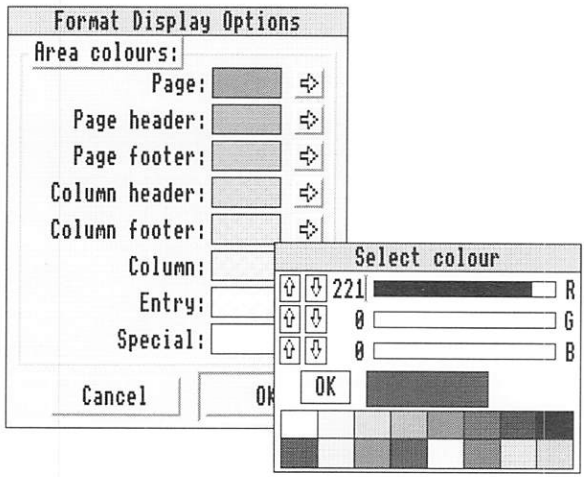
The first option in paper size is 'show'. Setting this option will display the paper margins of your printer driver on the screen as black. Your printer driver will need to be loaded in for this option to be effective.

Load in a *RISC OS printer driver* and set this option.

Next to 'show' is a popup menu displaying the list of standard paper sizes.

You have selected A4 in AFG and this is displayed here. S-Base provides an option for users with custom paper sizes. Setting this option will allow you to enter the dimensions of your paper's height and width in millimetres.

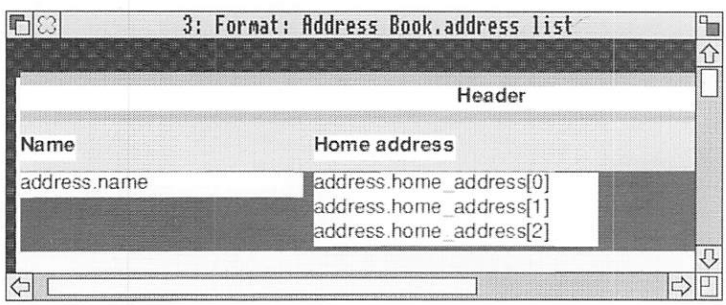
Click on the format display tool.



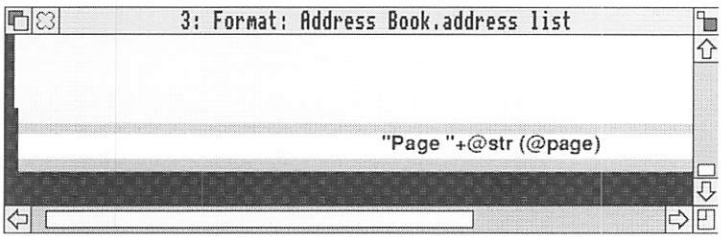
The format display dialogue allows you to set the colours used to represent each area of a report. This does not affect the prompts or fields making up your report. Remember: areas are not printed when the report is executed.

The current area colours are graduated grey scales. On some monitors it is very difficult to distinguish the column and entry colours.

Change the colour of the entry area to red. The changes to the area colours will not be actioned until you click 'OK' in the format display dialogue.



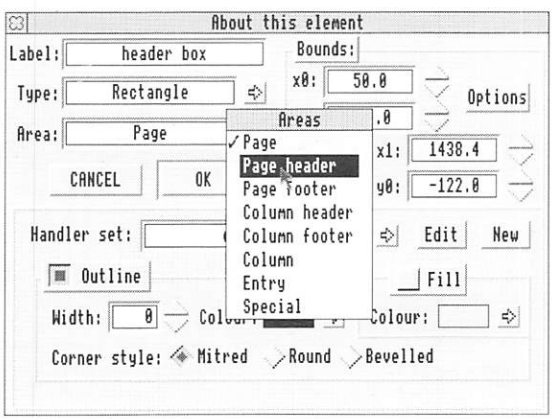
You can see the layout of the report and each of the areas. You will notice how each of the prompts and fields are completely enclosed by the area they are associated with. The black border around the entire report is the paper margins for the RISC OS printer driver.



You will notice that the margins of the printer driver used in this example overlaps the bottom of page. In the example above the page footer would not be printed when the report is executed. You will need to reposition the page footer area and prompt, and resize the bottom of the page. Alternately you can adjust the margin of your printer driver. If your printer driver is loaded when you start AFG, S-Base will automatically fit the report inside the page.

The page footer contains an expression to print the current page number. The **@page** function returns the current page number. This is converted to a string with the **@str** function. This is finally added to `"Page"` to produce the expression `"Page 1"` for the first page.

Double-click on the header prompt. Change the expression to `"Address List"` (include the quotes).

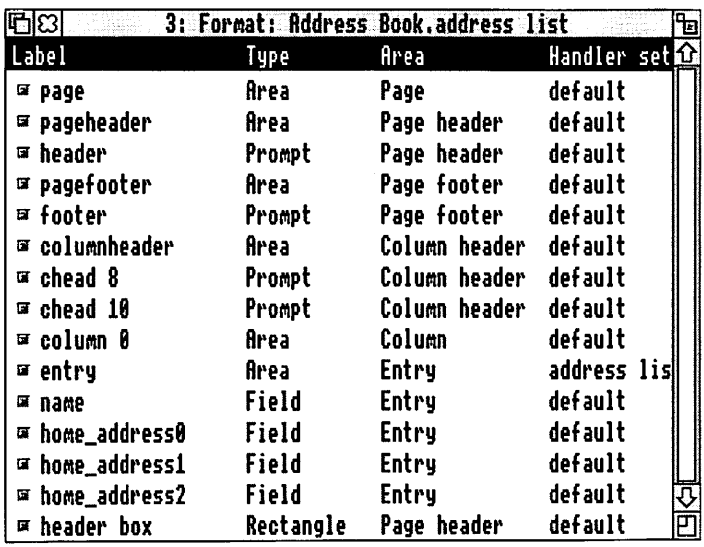


Select the rectangle tool and put a box around the page header labelled `header box`.

Click on the area popup and select 'page header' from the menu. The element area field must match the area it will be associated with otherwise it will not be displayed correctly. Switch the fill option off for the rectangle then click 'OK'. The bounding box of the rectangle must lie inside the page header area so resize it using Adjust if necessary.

Element ordering

The ordering of report elements is very important. S-Base insists on elements associated with the same area type being grouped together. The area element must always precede the elements contained within it. S-Base also insists that groups are arranged in the order that S-Base will use them. Open the list viewer for the `address list` format. This will show the order of the elements.



3: Format: Address Book.address list

Label	Type	Area	Handler set
page	Area	Page	default
pageheader	Area	Page header	default
header	Prompt	Page header	default
pagefooter	Area	Page footer	default
footer	Prompt	Page footer	default
columnheader	Area	Column header	default
chead 8	Prompt	Column header	default
chead 10	Prompt	Column header	default
column 0	Area	Column	default
entry	Area	Entry	address lis
name	Field	Entry	default
home_address0	Field	Entry	default
home_address1	Field	Entry	default
home_address2	Field	Entry	default
header box	Rectangle	Page header	default

The first element must always be the 'page area'. Everything is surrounded by this area so it must be first. Next is the 'page header' area followed by any elements enclosed by it. The 'page footer area' and its elements follow next.

The body of the report must be created next. The 'column header' area and

its elements are the first items to be used in the body of the report. Next is the 'column' area followed by the 'entry' area. This is the only element that uses a handler. The remaining elements are the contents of the entry.

AFG will create each group of elements in the correct order. If you decide to add extra elements to your report you should place them in the correct order. The `header box` element is at the bottom of the list. Position it between the `page header` area and `header` prompt. This will ensure it is printed behind the header. If the box was filled it would overwrite the page header if it was not in the correct order.

Within format layout you can used the 'bring to front' and 'put to back' tools to change the order of elements. Whenever a report does not produce what you expect you should suspect element ordering to be the problem.

Request entry handlers

A report consists of a number of entries flowing down the columns of a report. S-Base generates the entries one at a time by evaluating the expressions in the entry elements.

The request entry handler is the only handler required for a report. This handler is called every time S-Base wants to generate an entry. You can use **@entry** to find out which entry S-Base wants to generate. This returns the entry number, with the first being zero. The handler should load any data S-Base will need to generate that entry.

Create a new handler labelled `address entry` and attach it to the `address list` handler set. The handler should be of type 'S Request entry'. Enter the following code in this handler:

```
if @entry < @recordcount address
   load address,@entry+1
else
   report exit run rhandle
endif
```

This code checks to see if the entry number is less than the number of records in the `address` file. In this case **@entry** acts as a counter and is incremented when S-Base generates the next entry.

If the value of **@entry** is OK, the handler loads `address` record `@entry+1`. Remember that report entries are numbered from 0 whereas record numbers start at 1. To convert an entry number to a record number

you must add 1.

If S-Base is requesting a record which does not exist in the file, the **report exit run** command is executed. This command is the report version of **card exit poll** and behaves in an identical way. `rhandle` is the handle of the report.

S-Base may need to call your request entry handler a number of times for the same entry. For this reason it is important to use the value of **@entry** to calculate which record to load. Where this is not possible the function **@lastentry** is provided to check if S-Base is requesting the same entry twice. **@lastentry** returns the number of the last entry generated. This will be the same as **@entry** if the entry is being generated for a second time. Remember, S-Base can request the same entry as often as it likes, but will never request an earlier entry or skip an entry.

Running reports

Load the `startup` program and insert the following procedure definition below the `address_card` procedure:

```
def proc address_list
   local rhandle

   report new rhandle,"address list","Address List"
   report show rhandle
   report run rhandle
   report dispose rhandle
enddef
```

You will recognise this code. It is similar to that used for creating a card. The report is created with the **report new** command, shown with **report show**, and generated with **report run**. Finally **report dispose** disposes of the report when the user has finished with it.

The **report new** command has a report title argument appended to the end. This will appear in the title bar of the report window. It will not appear if the report is sent to the printer. If you require the report to be output to the printer instead of the screen substitute the **report show** command with **report print**. Pressing Esc will terminate the report while it is printing.

Replace the **error** command at the bottom of the `startup` program with this code to call the `address_list` procedure:

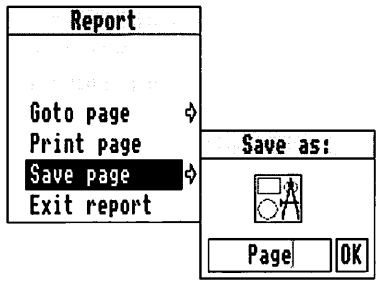
```
...
```

```
      proc address_card
   when 1
      proc address_list
   when 2
      quit = TRUE
...
```

Try running your application and selecting the `address list` option. The report should run and you will see a list of names and addresses.

Controlling reports

Click Menu over the report to open the report menu. This is automatically generated and controlled by S-Base.



'Next page' and 'Previous page' allow you to move between pages of the report. These options are equivalent to pressing Page Up and Page Down on the keyboard. You can see that neither is available in this report since it is only one page long. You can add more records to the `address` file to get these options to work.

'Goto page' allows you to specify a page to go to. S-Base will not allow you to go to a page that does not exist yet. If you have only viewed the first five pages of the report the 'Goto page' option will not accept pages beyond page five.

'Print page' will send the report page to the RISC OS printer driver. The report will be printed with graphics if it is a normal report or with text for a text-only report (see below). Try this with your sample report.

'Save page' will save the report page to another application or file. A normal report will be saved as a draw file with a text-only report being saved as a text file.

The last option will exit from the report and is equivalent to pressing Esc.

Text-only reports

Some printers take a long time to print a report such as you have designed here. For this reason, S-Base supports text-only reports. The restrictions on these are explained under **report new** in the S Reference. Basically, everything must be aligned with the text-only grid, you cannot print graphics and only system font is allowed.

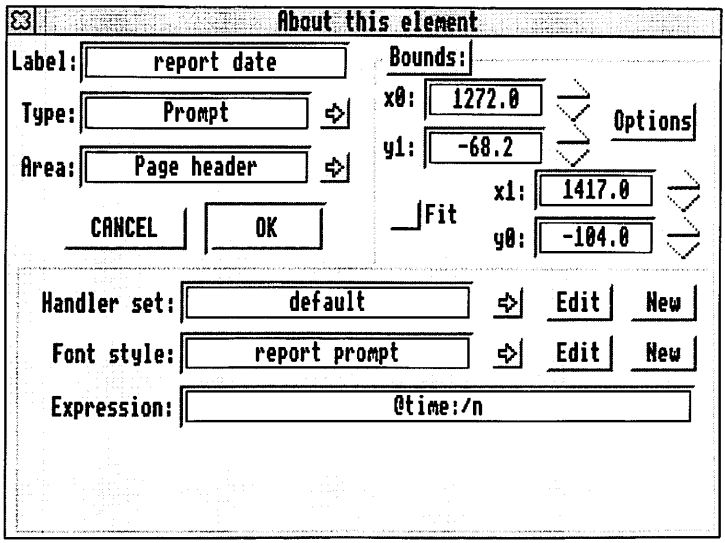
The advantage with text-only reports is that they can be generated very quickly. Some dot matrix printers can print a page in 20 seconds if it is text, but take several minutes if the page contains graphics.

A report can be marked as text-only by setting the option in the format info dialogue for the report's format. The report cannot contain any lines, rectangles or fancy fonts.

You can try generating a format for a text-only report. In AFG select the `system` font style for everything and generate a new format. Then open the Grid dialogue and select the text-only grid (with 'Show' and 'Lock' on). Select all the elements in the format and snap them to the grid. Then set the format to be text-only in the Format info dialogue. You will have to alter the **report new** command in the `startup` program to use this new format. Run your application and try printing the report. On a dot matrix printer it will be produced much more quickly.

Formatting expressions

The final addition to your report is a prompt to identify when your report was printed. Display the `address list` report in the format layout editor. Select the prompt tool and create a prompt labelled `report date` just inside the right edge of the page header.



Change the horizontal alignment option to 'right' and the area to 'Page header'. Use the `report prompt` font style. The expression should be `@time:/n`.

The **@time** function returns the current time. This function will be discussed in the next chapter. The remainder of the expression is formatting information.

Run the `address list` report and examine the results. You will see the time has been formatted to appear as a date.

Expression formatting is commonly used on dates, times, and numeric data. Dates in particular need to be input and output in a variety of different formats.

A colon separates the data from the formatting information. Formatting information consists of conversion codes, alignment and width information and finally a number of decimal places or maximum string length. Each is separated by a colon and is optional.

Conversion codes always begin with a slash. The codes are listed in the S Reference under **print**.

Alignment and width information is given as a positive or negative number of characters. Positive means right-aligned and negative means left-

aligned. Centre alignment is not supported.

Try this example on the command line (you will not get the same output):

```
s:print @time:/n:15
       27/08/92
```

This means: take the time, extract the date in 'natural' format (the `:/n`) and right-align it in a field of 15 spaces (the `:15`), then print the result.

For floats you can give the number of decimal places to use. Try these examples on the command line:

```
s:a = 1234.4321
s:print a:-10:2
1234.43
s:print a:10:3
  1234.432
s:print a:10:0
      1234
```

At this point in the tutorial you should have sufficient skills to develop a fairly comprehensive application. The next few chapters will examine a range of S-Base functions you will find useful for processing data. Following that you will learn about some of the more advanced areas of S-Base. It is recommended that you study all the material to date and ensure you have a good understanding of the fundamentals of S-Base.

16. Dates and times

Recording dates and events is an everyday operation for most databases. These can range from birth dates and ages to the date stamp on an invoice transaction. This chapter will explain the many date and time functions available in S-Base. You will extend your address book application to store birth dates and calculate an up-to-the-minute age. Consult the S Reference for full descriptions and examples of functions listed in this chapter.

Date and time functions

S-Base has nine dedicated date and time functions. These are listed here:

@date	returns the current system date. There are no arguments.
@todate	converts an integer or time argument into a date.
@datesn	converts a string argument into a natural format date.
@datesi	converts a string argument into a interval format date.
@time	returns the current system time. There are no arguments.
@totime	converts an integer or date argument into a time.
@timesn	converts a string argument into a natural format time.
@timest	converts a string argument into a time format time.
@timesi	converts a string argument into a interval format time.

In addition to these dedicated functions there are a number of generic functions. These process a time or date and return a value of the same type as the argument. If the argument is a date then a date is returned. If the argument is a time then a time is returned.

@times	converts an argument into an integer representing the seconds component of the argument. This is of little use for dates as it always returns 0.
@timeu	converts an argument into an integer representing the minutes component of the argument. This is of little use for dates as it always returns 0.
@timeh	converts an argument into an integer representing the hours component of the argument. This is of little use for dates as it always returns 0.
@timedow	converts an argument into an integer representing the day

of the week.

@timedom converts an argument into an integer representing the day of the month.

@timedoy converts an argument into an integer representing the day of the year.

@timemoy converts an argument into an integer representing the month of the year.

@timedm converts an argument into a string representing the day and month.

@timemy converts an argument into a string representing the month and year.

@timey converts an argument into an integer representing the year component of the argument.

@timefod returns the time/date at the start of the day from an argument. This is of little use for dates.

@timefom returns the time/date on the first day of the month from an argument.

@timefoy returns the time/date on the first day of the year from an argument.

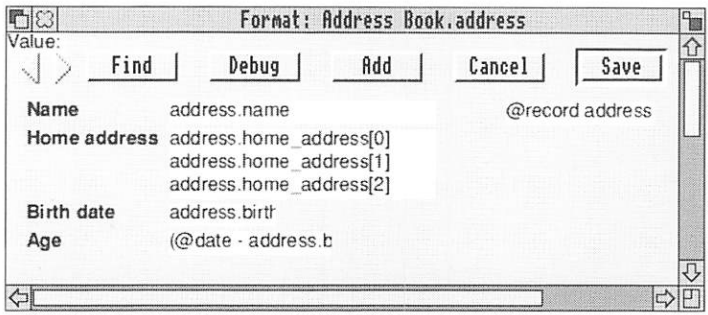
Date and time arithmetic

S-Base stores dates and times internally using a format similar to integers. Dates are stored in intervals of days. Times are stored in intervals of seconds. Both types support null dates and times. These will appear as spaces when displayed. Attempting to assign an invalid date or time to a variable or file field will result in a null value being used instead. Arithmetic operations performed on dates and times will have the same effect as arithmetic operations on integers.

Try the following examples on the command line:

```
s:a = @date
s:print a
05/09/1992
s:print a + @datesi "1d"
06/09/1992
s:b = @totime a
s:b = b - @timesi "1s"
s:print b
04/9/1992,23:59:59
```

You will now extend your address book application and add birth date and age fields. Add a new file field to the `address` file labelled `birth_date`. You will need to create a new data type labelled `date`. This will be of type 'Date'. Load the `address` card into the format layout editor. Create a prompt containing the expression `"Birth date"` and position it under the home address prompt.



Create a field labelled `birth date` and position it next to the prompt. The field should use the `address fields` handler set and the expression will contain `address.birth_date:/m`. The field will be formatted into month format. Create a prompt containing the expression `"Age"`. Position it below the birth date prompt. Create a field labelled `age` and position it next to the prompt. The options for the field will have automatic recalc set off and protect set on. The expression will contain `(@date - address.birth_date):/i`. To calculate an age you will need to subtract the birth date from the current date. The result should be displayed in interval format.

Create an S Field Exit handler labelled `birth exit` and attach it to the `address fields` handler set. The S Field Exit handler is called when the cursor leaves the field. It allows you to update the field or other fields related to it. Enter the following code into the handler:

```
if @elementlabel = "birth date" then card update
ahandle,"age"
```

This handler will update the age field after the birth date has been altered. Try running your application. S-Base will notify you that the structure of the database has been altered. Click on 'Import' to import your existing data under the new data structure. Try entering some birth dates such as '`1jan72`'.

17. Mathematics functions

Mathematical functions in S-Base can be split into a number of sections. The S language provides functions for trigonometry, statistics, number conversion and general mathematics. This chapter will examine each of these individually.

Consult the S Reference for full descriptions and examples of functions listed in this chapter.

Trigonometry functions

The following trigonometry functions are provided by S-Base:

@acos	calculates the arc cosine of a number and returns a float value in radians.
@asin	calculates the arc sine of a number and returns a float value in radians.
@atan	calculates the arc tangent of a number and returns a float value in radians.
@atanq	calculates the arc tangent of two numbers y/x, returning a value in radians.
@cos	calculates the cosine of a radian number and returns a float value.
@cosh	calculates the hyperbolic cosine of a radian number and returns a float value.
@exp	calculates the exponent of a number and returns a float value e ^{number} where e is the constant 2.718281828.
@ln	calculates the logarithm of a number to base e where e is the constant 2.718281828.
@log	calculates the logarithm of a number to base 10.
@sin	calculates the sine of a radian number and returns a float value.
@sinh	calculates the hyperbolic sine of a radian number and returns a float value.
@tan	calculates the tangent of a radian number and returns a float value.

@tanh calculate the hyperbolic tangent of a radian number and returns a float value.

Statistical functions

The following statistical functions are provided by S-Base:

@max finds the maximum of a set of numbers. The numbers are passed as arguments.

@mean calculates the mean of an array of numbers. The first argument is an array variable. If a subscript is given the calculations begin from that element. The second argument is the number of elements to use. This argument is optional with the default being all.

@min finds the minimum of a set of numbers. The numbers are passed as arguments.

@sdev calculates the standard deviation of an array of numbers. The first argument is an array variable. If a subscript is given the calculations begin from that element. The second argument is the number of elements to use. This argument is optional with the default being all.

@sum calculates the sum of an array of numbers. The first argument is an array variable. If a subscript is given the calculations begin from that element. The second argument is the number of elements to use. This argument is optional with the default being all.

Number conversion functions

The following number conversion functions are provided by S-Base:

@abs converts a number into a positive signed value.

@deg converts a number in radians into an equivalent value in degrees.

@dollar truncates a float to two decimal places. This function is used with currency values.

@int converts a float number into an integer value.

@rad converts a number in degrees into an equivalent value in radians.

@round rounds a float number to a specified number of decimal places. The first argument is the number with the second being the number of decimal places.

General mathematics functions

The following general mathematics functions are provided by S-Base:

@pi returns the value of PI (3.1415926536).

@rnd returns an integer random number. The argument specifies an upper bound for the result. Numbers generated will be 0 to (bound – 1).

@sgn calculates the sign of a number and returns a signed integer. Negative = -1, zero = 0, positive = 1.

@sqr calculates the square root of a number and returns a float value.

18. Strings

This chapter lists the facilities S-Base provides for the manipulation and processing of string values. You will add a comment field to your `address` card and learn how S-Base handles large string elements.

Consult the S Reference for full descriptions and examples of functions listed in this chapter.

The basics

S-Base implements two methods of string storage. The first is for string variables and is very flexible while the second is for string file fields and is rather rigid.

The length of both string variables and file fields can not exceed 254 characters. String file fields are padded with spaces to a fixed length and can only contain characters specified within the upper and lower limits in the data type.

String variables are null (ASCII 0) terminated and empty when initialised. They can store any ASCII character except null. String variables can expand or contract, up to the maximum size of a string.

String variables can be concatenated (joined) together by adding them in an equation.

Try the following on the command line:

```
s:a = "hello"
s:b = "there"
s:print a + " " + b
hello there
```

Remember to ensure the concatenated length of two strings does not exceed 254 characters. Concatenating file fields is slightly different. Because they are padded to a fixed length it is important to remove the trailing spaces with **@trim**.

For example,

```
s:address.name = "Steve"
s:print address.name + "Simon"
Steve                Simon
s:print @trim address.name + "Simon"
SteveSimon
```

String functions

The S functions relating to strings are listed below. For more information you should consult the S Reference.

@asc converts a single string character into an equivalent ASCII code. The result returned is an integer.

@chr converts an ASCII code (integer 0..255) into an equivalent string character.

@formstr formats a string so that it can be divided into set lengths without breaking a word. The first argument contains the string with the second containing the format width. For example, `@formstr("This is just a test",9)` would format the string with extra spaces to produce `"This is just a  test"`. This can be chopped into lengths of 9 characters without breaking a word.

@instr finds occurrences of a substring within another string. The first argument is the string to be searched. The second argument is the substring to search for. The third (optional) argument is the position in the string to search from. The default is 1.

@left extracts the left part of a string. The first argument is the string. The second argument is the number of characters to extract.

@len returns the length of a string.

@match checks a string for a match against a wild card substring. The first argument is the string. The second argument is a string containing wild cards. The wild cards are: ? to match any character and * to match any string. The result will be false (0) for failure, true (1) for success. For example, `"Hello*"` will match any string starting with `"Hello"`.

@mid extracts a number of characters from a string. The first argument is the string. The second argument is the start position. The third argument is the number of characters to extract.

@msg translates a tag string into a language string. The tag string looks up the 'message' file inside your

application's directory. If it finds a match for the tag string it returns the translated string.

@mult returns a string whose length is a multiple of a given length. Padding spaces are added if necessary. The first argument contains the string. The second argument contains the multiple value.

@onespace returns a string containing a maximum of one space between words.

@pad returns a string padded with characters to a specified length. The first argument is the string. The second argument is the desired length of the string. If the string is greater than this length it will be truncated. The third argument is the character to pad with. This is optional with the default being space.

@right extracts the right part of a string. The first argument is the string. The second argument is the number of characters from the right to extract.

@spc returns a string containing spaces, the length of which is given by the argument.

@string returns a string containing multiple copies of a substring. The first argument contains the number of copies. The second argument contains the substring.

@stringize converts an array of ASCII characters into a string. The first argument is an array variable. If a subscript is given the conversion will begin from that element. The second argument is the number of characters to convert. This function is useful for extracting strings from operating system data blocks.

@tolower converts a string to lower case.

@toupper converts a string to upper case.

@trim trims all trailing spaces from a string.

@val converts a string containing digits into a numeric value.

@str

S provides a universal command to convert any of the base data types into a string. **@str** uses the formatting information to convert numeric, date and

time data into an equivalent string.

Try some of these examples on the command line:

```
s:a = @timesn "1/4/92"
s:print @str(a:/t)
01/04/92,00:00:00
s:a = 1234.5678
s:print @str(a:9:2)
  1234.57
```

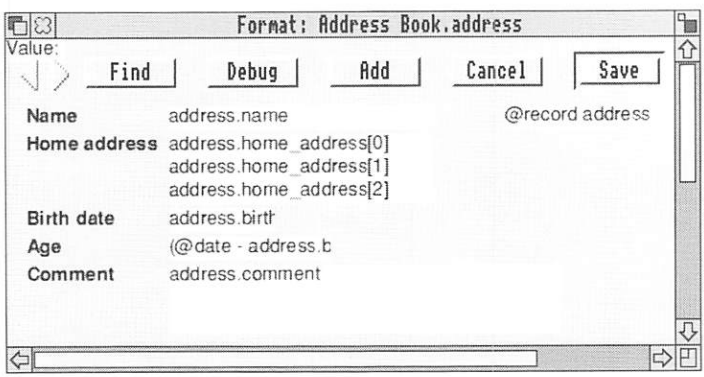
When converting data into strings you should take into account preceding and trailing spaces. If you use a width of 10 characters and there are only 5 digits S-Base will pad the string with spaces.

Strings in formats

String fields in formats are treated slightly differently from other types of data. String fields tend to be larger than numbers and times, and need to be formatted as they are typed. If the text in a field is wider than the field bounds, S-Base will automatically scroll the field. This is also true with numbers and times though strings can make use of multi-line fields.

Add a file field labelled `comment` to the `address` file. You will need to create a new data type labelled `comment`. This will be a string of 120 characters. It will have a lower limit of 32 and no upper limit.

Create a prompt in the `address` card with the expression `"Comment"` and position it below the `age` prompt.



Create a field labelled `comment` and position it below the `age` field. The field will contain the expression `address.comment` and use the `address`

`fields` handler set. Drag out the bottom right corner of the field to allow enough room for 3 lines of text. This is called a *multi-line* field. Text will flow from one line of a multi-line field to another. Multi-line fields word-wrap so words will not be broken. If the field size is not big enough to accommodate all the text in the field it will scroll vertically. The field will not scroll in the horizontal direction.

Run your application and enter a long comment into a record.

This is the last chapter to cover the basics of S-Base. The remainder of this tutorial will cover more advanced features of S-Base.

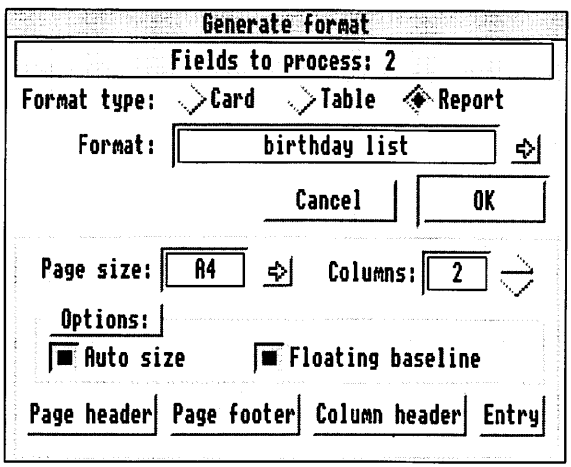
19. Advanced reports

You will already have a good understanding of reports and should be able to produce very professional results. This chapter will look at some advanced features of the S-Base report system. You will modify your `address` list report to print more than one format at a time.

From this chapter onwards the tutorial will deal with overall concepts rather than individual instructions. This tutorial will only explain new commands and functions. It is assumed you have a good understanding of all the material covered so far.

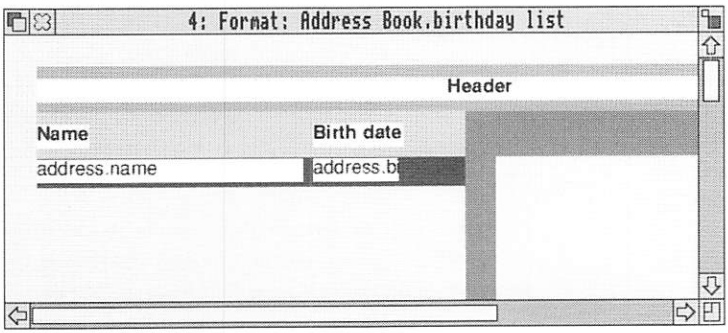
Multi-column reports

Create a blank handler set labelled `birthday list`. Open the `address` file field list and select the `name` and `birth_date` file fields. Use AFG to generate a report format labelled `birthday list`.



Set the number of columns to two. This will create a report with two columns (multi-column). There is a limit of 99 columns for a report though no practical report would ever contain this many columns.

Set the settings at the bottom of the dialogue to the same values as the `address list` report. Set the handler set for the entry to `birthday list`. Click 'OK' and generate the report.



The report layout will be similar to the `address` list report. On the right side of the report you will see the extra column. S-Base will automatically create a gap between the columns to enhance the layout. There is no column header created for the extra column. S-Base generates this when the report is executed.

Double-click on the header prompt and change the expression to `"Birthday List"`. You may also wish to add the current date to the top right of the report (you can drag this from the format layout editor of the `address list` report if you like).

Double-click on the birth date field (shown as `address.birth_date`). Add formatting information to print the birth date in month format (:/m).

S-Base will not add formatting information in AFG since it does not know which format to use. If a field in a card or a report is chopped in half or incomplete it will be because there is no formatting information. This is the most common error encountered with formats.

This report will print a name and birth date down the first column and flow onto the next column when it reaches the bottom. If you have used a Desktop Publishing package you will be familiar with this feature. The `address` file may not contain enough records to fill up a column yet. It may be necessary to add more records or reduce the length of the columns.

Select both columns and drag the bottom edge of the columns to quarter of their original height. You will need to drag them back to their full size after you have tried out this report.

In the handler list viewer, copy the `address entry` handler and label it `birthday entry`. Attach the `birthday entry` handler as an S Request

Entry handler to the `birthday list` handler set.

An alternative would have been to attach the `address entry` handler to the `birthday list` handler set. However, this handler will be altered substantially later in this tutorial and a little foresight now has saved you extra work in the future.

You will find that many reports will use the same request entry handler. The majority of reports only require an request entry handler to load a record or two with all the variation in the report occurring in the format. This is also true for the calling procedure in your `startup` program.

Alter the `startup` program to match the following (leave the top part alone):

```
def proc address_list(fmt)
   local rhandle
   report new rhandle,fmt
   report show rhandle
   report run rhandle
   report dispose rhandle
enddef

| Create the main menu

pmenu new mhandle,"Address Book",1,4+8+64
pmenu insert mhandle,"","1","Edit address data"
pmenu insert mhandle,"","2","Print address list"
pmenu insert mhandle,"","3","Print birthday list",1
pmenu insert mhandle,"","4","Exit system"

| Prompt the user then do something about it

quit = FALSE

repeat
   pmenu show mhandle
   pmenu poll mhandle,option
   pmenu hide mhandle
   case option
   when 0
      proc address_card
   when 1
      proc address_list("address list")
   when 2
      proc address_list("birthday list")
   when 3
      quit = TRUE
   endcase
until quit
```

```
pmenu dispose mhandle
```

The `address_list` procedure has been changed so that the name of the report format is passed in as a parameter. This allows the same procedure to be used for both the `address list` and the `birthday list` reports. An extra option has been added to the main menu and its corresponding case structure at the bottom of the code.

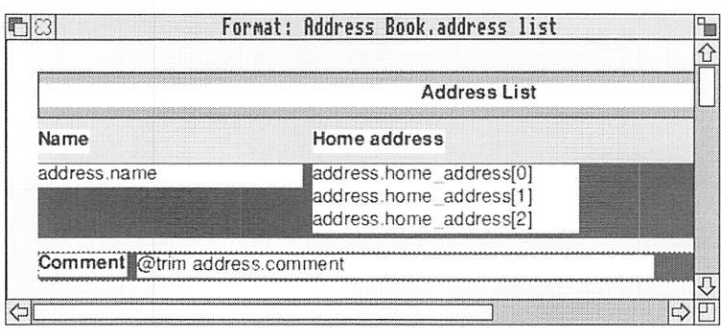
Run your application and try the `birthday list` report.

Multi-entry reports

Multi-entry reports are also available in the S-Base reporting system. Sometimes it is necessary to change the report layout while it is being executed. An example of this can be seen in the Video Library application. The `Video Catalogue` report has an option to display varying amounts of information in the report. If full information is requested a second entry type is also displayed.

You will use this feature with the `comment` file field in the `address` file. This field takes a considerable amount of space to print. It would be more efficient to print the `comment` field only if it contained text. That would mean changing the entry for the records that contain text in the `comment` field.

Examine the `address list` report in the format layout editor. Create another entry area (or copy the old one) and label it `second entry`. Position it below the `address` entry area. It can be as far down the page as convenient provided it is enclosed by the column area. S-Base will position it directly below the first entry area when the report is executed.



Add a prompt labelled `prompt 1` with the expression `"Comment"`. It

should be in the 'entry' area and use the `report prompt` font style. Create a field labelled `comment` with the expression `@trim address.comment`. It will also be in the 'entry' area but will use the field font style. Set the floating baseline option on.

Position it to the right of the `comment` prompt and drag out the field width to a reasonable size. When generating the report, S-Base will extend the baseline of the comment field to fit the text within it.

Load the `startup` program and alter the first line of the `address_list` procedure definition to match the following:

```
local rhandle, recnum=1
```

This line declares a new local variable `recnum` and initialises it to `1`. Load the `address entry` handler and alter it to match the following code:

```
if recnum <= @recordcount address
   load address,recnum
   if @trim address.comment <> ""
      report set entry rhandle,"second entry"
   else
      if @entry <> @lastentry then recnum += 1
   endif
else
   report exit run rhandle
endif
```

The entry handler will check to see if all the records have been printed yet. If so, the report will exit. If there are more records to print it will load the next `address` record.

Next it will check to see if the comment field contains any text. If it does S-Base will now use the `comment` entry area. The **report set entry** command has two arguments. The first argument is the report handle. The second argument is the label of the new entry area. S-Base will use this entry area until instructed otherwise.

If the record does not contain comment text the handler checks whether S-Base is calling this entry for a second time. **@entry** and **@lastentry** will be identical if the entry is being called again. S-Base usually does this when it has reached the bottom of a column and needs to flow an entry to the next column. The handler will increment `recnum` by 1 if this is not a repeated entry.

Create a new handler set labelled `comment` and attach a new request entry

handler labelled `comment entry`. Enter the following code into the handler:

```
if @entry <> @lastentry then recnum += 1
report set entry rhandle
```

This handler will check to see if this is a repeated entry and increment `recnum` if not. Next it will tell S-Base to use the original entry area for the next entry. If no entry area label is supplied S-Base assumes it is the original entry area. S-Base will always use the first entry area it encounters in the format element list when the report is executed. Remember to check the order of the elements in this list before executing your application.

Attach the `comment` handler set to the second entry area. Run your application and test the report.

The size of each entry will differ depending on the amount of command text to display. The floating baseline option will automatically resize the field and entry area to fit the text.

20. Handlers and handler sets

This chapter will give an in-depth look at handlers and handler sets. Each type of handler will be explained with examples. The creation and editing of handlers and handler sets has been discussed in earlier chapters of this tutorial. This chapter will not revisit these areas. It is recommended that you consult examples given in earlier chapters to help your understanding of this area of S-Base.

S-Base handlers

S Click

This handler is called when the user clicks Select or Adjust on an element. The handler should check which element has been clicked on and take the appropriate action.

S Keypress

This handler is called when the user presses a key from within a format field. The handler can check the current value of the field and search for matches in the database.

The keypress handler is called after the key has been processed by S-Base and before the field has been updated. The handler should not use the value of the field as this will be out of date. It should use the buffer value which can be found with **@elementvalue**.

S Request entry

This handler is called when S-Base wants to generate an entry of a table or report. The handler should load any data required to generate the entry.

S Enter entry

This handler is called when the cursor enters a new entry. The handler can prepare the entry for the user to modify. This may involve updating cards associated with displaying the current entry.

S Exit entry

This handler is called when the cursor exits an entry. The handler can check to see if the entry has been altered. If so, it may decide to save it to disc.

Once this handler is called you should not attempt to modify the entry

you are about to enter. S-Base will have already decided where the cursor is going. For example, you cannot protect elements in the exit handler, since if the user has clicked in one of those elements, S-Base will still place the cursor in it. This may result in the cursor appearing in a protected entry. These operations should be performed in a keypress handler.

You should not attempt to alter the cursor's location in an exit handler, or S-Base may become confused.

S Extension

This handler is called when the user attempts to extend (add more entries to) a table. The handler must decide whether the table can be extended and by how much. If so, it should set up the new entries ready for S-Base to use.

Variables containing table handles cannot be used in this handler if the table contains zero entries and is being initialised. The **table new** command will attempt to create an entry for the table. The variable will not have been assigned the table handle at this point because the **table new** command is still being executed. Use **@handle** to return the handle of the table.

S Enter field

This handler is called when the cursor enters a format field. The handler can set up any data necessary to display or manage the field.

S Exit field

This handler is called when the cursor exits a format field. The handler can update the field or process the data in the field.

Once this handler is called you should not attempt to modify the field you are about to enter. S-Base will have already decided where the cursor is going and any change to the fields may result in the cursor appearing in a protected field. These operations should be performed in a keypress handler.

You should not attempt to alter the cursor's location in an exit handler, or S-Base may become confused.

S Menu maker

This handler is called when the user clicks Menu over a card or table. The handler should create and display a menu. S-Base will open the

menu at the current mouse pointer position.

S Menu select

This handler is called when the user selects a menu item. The handler should work out which item was selected and take appropriate action.

S Redraw

This handler is called after S-Base has finished redrawing a card. The handler can use the graphics commands to add drawings to the card.

Do not attempt to draw outside the bounds of the card. This may interfere with other applications currently active in the desktop environment.

As a general rule handlers should never attempt to dispose of the object that has called them. S-Base will attempt to find a calling object that no longer exists and will become confused once it has exited the handler.

When S-Base calls a handler it will set up the *current object*. The current object is the object which caused the handler to be called. The current object is always a card, table or report. Many S functions will assume the current object if you do not specify an object handle.

Handler functions

There are a number of functions which S-Base sets up before calling a handler. These functions are used by the handler to decide how best to process the event.

@altered	returns whether a card or table has been altered by the user.
@clickelement	returns the number of the element the user has clicked on. This function should only be used in the click, menu create and menu select handlers.
@clickelementlabel	returns the label of the element the user has clicked on. This function should only be used in the click, menu create and menu select handlers.
@clickmouse	returns the state of the mouse buttons when the element was clicked on.
@cursor	returns the current offset of the cursor into a field.
@elaltered	returns whether the element has been altered by the

user.

@element	returns the number of the element the user has pressed a key in. This function can be used in all handlers except the click, menu create and menu select handlers.
@elementlabel	returns the label of the element the user has pressed a key in. This function can be used in all handlers except the click, menu create and menu select handlers.
@elementvalue	returns the value of the data entry buffer for the element.
@entry	returns the current entry number.
@entryaltered	returns whether an entry has been altered by the user.
@handle	returns the handle of the current object.
@justaltered	returns whether the keypress has just altered the field. This function should only be used in the keypress handler.
@key	returns the ASCII code of the key pressed. This function should only be used in the keypress handler.
@lastentry	returns the number of the last entry called.
@menu	returns the handle of the current menu. This function should only be used in the menu select handler.
@menufind	returns the handle of a given menu. It will return –1 if the menu does not exist.
@menuselect	returns the item number selected in the menu. This function should only be used in the menu select handler.

The default event handler

S-Base provides a special handler set labelled `default`. If the normal handler set does not have a handler for a particular event type, it will be passed the default handler set. Events which are not associated with a particular object can be trapped by this handler set. This feature is useful for trapping certain types of raw events.

Raw event handlers

S-Base provides raw event handlers for your application to directly interface with the operating system. If a raw event handler is present S-Base will pass the event to the handler instead of processing it itself. The event handler may decide to process the event, in which case it should return true (1), or leave the event to S-Base to process, in which case it should return false (0). If the handler returns true (1) S-Base will ignore the event. For example, a raw mouse click event would not be processed and S-Base would not subsequently call the S Click handler with the event.

The following S commands and functions are available to help the processing of raw events:

@eventtype	Returns the event type.
@addr (event)	Returns the address of the 256-byte event record.
load event	Loads the event into a 64 element integer array for processing.

Raw event handlers attached to a format will only be called if the event is related to that format. Events not related to a format will be sent to the `default` handler set for processing.

Processing raw events can be a complicated task and is beyond the scope of this tutorial. A full description of the raw events supported by the RISC OS Window Manager can be found in the RISC OS Programmers' Reference Manual.

21. Advanced Cards

This chapter covers the remaining aspects of S-Base cards. You will need to consult the Address4 application from the tutorial directory of your S-Base work discs. In this chapter you will have a detailed look at selectors and be shown examples of their use. You will also examine multiple cards running from a single **card poll** command.

Selectors

In simplified terms a selector is a special type of menu that is embedded into a card. It does not go away when it is clicked on and it can allow several items to be selected at once.

Load the Address4 application. There are a substantial number of additions to this application so you will have to look at them one at a time. Load the `startup` program into a text editor.

There will be a number of new options in the main menu. Ignore these and concentrate on the `View address data` option. This option calls a procedure `view_address`. This procedure first initialises a number of variables. The `name` variable is used to store the item that has been selected. It is initialised to 0 so that S-Base knows which item in the selector to select when the card is opened; 0 will select the first item. Secondly, it checks to see if there is anything to display. If there isn't, it will display an appropriate error message.

The procedure next creates the card. The card will contain a blank selector with no items. It is necessary to insert items into the selector before you can use it. This procedure inserts a new item for each `address` record on the database.

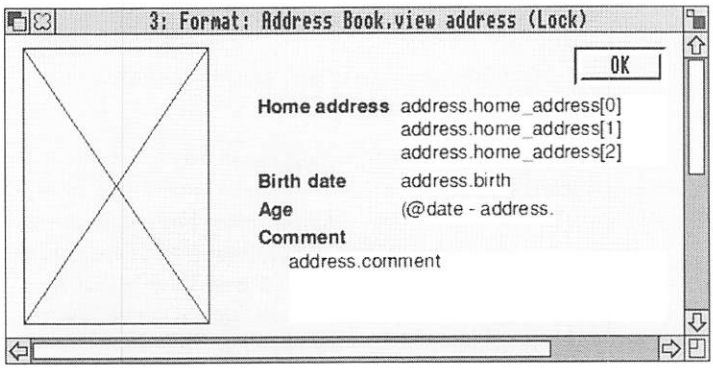
Selectors are a special type of element. They consist of a menu object attached to a selector type element. This is similar to the way icon elements have template icons attached to them. When you alter a selector it is necessary to use its object handle. The **@elementhandle** function returns the object handle of an element. Most element types do not have object handles and −1 is returned. The card handle and the element label need to be passed as arguments.

The **selector insert** command inserts items into a selector at a specified position. This command has three arguments. The first argument is the object handle. The second argument is optional and contains the item

number to insert before (the last item by default). The final argument is the text to insert into the selector. Once the selector has been set up the card is opened and polled.

Finally, the card is shown, polled and then disposed. It is necessary to load the first `address` record to set up the card fields. S-Base initially sets the field values when the card is created, not when it is shown. At this point the `address` record in memory could be anything. The **card update** command will update the data entry buffers to match the `address` record in memory.

Load the `view address` card into the format layout editor.

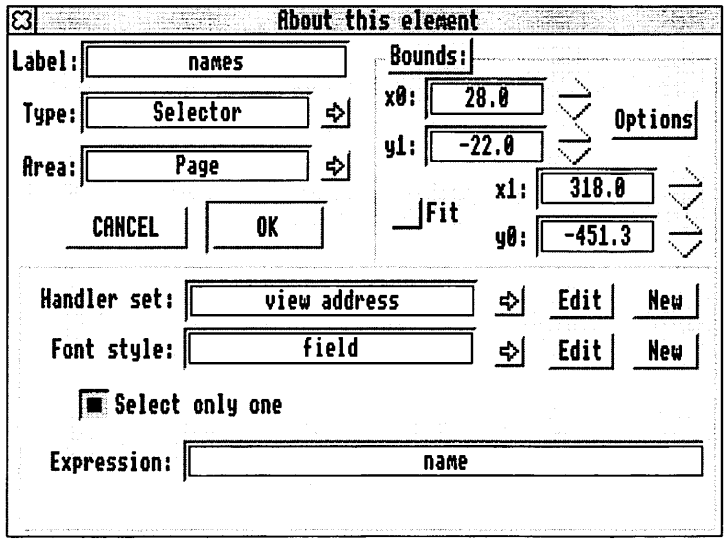


You will see a remodelled version of the `address` card. There is a single `OK` icon at the top right corner of the window. This will close the card when you have finished viewing addresses.

The left side of the card is occupied by a selector labelled `names`. The selector is designated by a box with crossed lines through it.

Double-click on the selector to view the element information.

The selector uses the `field` font style. This is the style that each of the selector's elements will be displayed with. The 'select only one' option has been set on. This option allows only one item in the selector to be selected at once. The item number is stored in the variable found in the expression field.



When the user clicks on a selector item S-Base will automatically update the `name` variable with the new selection number. Changing the value of the `name` variable and performing a **card update** command on the selector will cause the selected item to match the `name` variable.

Examine the `view address` handler set. It contains a single S Click handler labelled `view click`. This handler will exit the card if 'OK' is clicked on. When the selector is clicked on the handler will load a new `address` record and update the card. The item numbers of a selector begin at 0 so it is necessary to add 1 to align them to record numbers.

The contents of a selector can be added to, deleted from, or altered at any stage. This tutorial will not venture into these areas of selectors. For further details consult the S Reference.

Run the application and select the `View address` option. Examine the code to gain an understanding of the interaction between the selector and the handler.

Selecting multiple items

The other feature of selectors is their ability to contain a selection of several items. The process of selecting several items is performed in an identical manner to the RISC OS 3 filer. Clicking Select on an item will

select it and deselect the remainder. Clicking Adjust on an item will add or remove it from the selection. Selections can be created or added to by dragging with Select or Adjust respectively.

Load the `startup` program into a text editor and examine the procedure relating to the `Print selected addresses` option in the main menu.

This is similar to the previous procedure you examined, with a few exceptions. The variable labelled `number` contains the number of addresses to be printed. This is initialised to 0. The other new variable labelled `bitsel_handle` contains the handle of the selector's *bitstring*.

The bitstring of a selector contains the state of each item in the selector. S-Base uses the bitstring information to test if a selector item has been selected. The **@selectedhandle** function returns the bitstring handle of a selector. The function requires the object handle of the selector to be passed as the argument.

Load the `selected addresses` format into the format layout editor. This format consists of `cancel` and `print` icons, a field and prompt representing the number of addresses to print, and a selector. Double-click on the selector. You will notice the 'select only one' option is off. This selector can have any number of items selected. It can even have none selected.

Examine the handler set for the selector and the icons.

The `selected addresses` handler set is used by this card and the report this card generates. Both the card and report are related to each other and use different event types so it is OK to combine the two into one handler set.

Double-click on the selector with Ctrl and Shift held down to load the selector's S Click handler to into the text editor.

The handler consists of a case structure for checking the mouse click and a locally declared procedure. When the `cancel` icon is clicked on, the card exits. When the selector (`names`) is clicked on, a value is assigned to `number` and the field is updated on the card.

The **@bitsetcount** function returns the number of items that are selected in the bitstring. The function requires the bitstring handle to be passed as the argument.

When the `print` icon is clicked on, the handler hides the card. The **card hide** command temporarily closes the card. The card has not been deleted and can be displayed with a **card show** command.

Next, the handler calls the procedure `print_away`. This procedure dimensions a local array to hold all the record numbers being printed, then tests each item of the selector to see if it is selected. The **@bittest** function checks the bitstring and returns true (1) if the item is selected. The procedure stores the selected items in the `recs` array and increments `numrecs` by 1. After it has checked all the items in the selector it creates the `selected report` report and runs it. The card exits after the report has been run.

This report prints a simple name, address and date of birth. Close the text editor with Adjust. This will open the handler set which is the parent of the window. Load the request entry handler. The handler checks to see if all the records have been printed and exits if so. Otherwise it loads the next `address` record for the report.

Run your application and select a number of items. Experiment with the selection system on the selector and print some names.

Concurrent polling

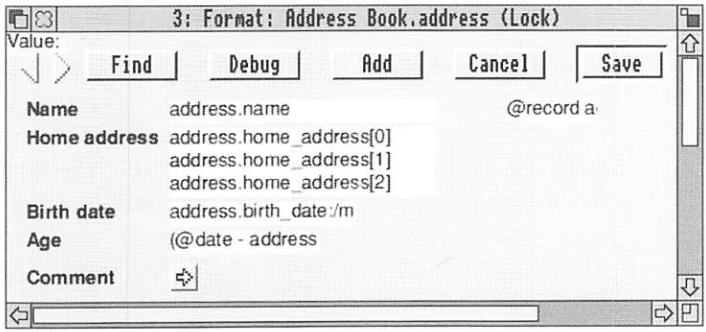
So far you have looked at nested polling where a card polls inside another **card poll** command. This can be advantageous if you are using scared cards though becomes very cumbersome if you are displaying a record in a number of cards simultaneously.

Address4 contains a modified `address` card. The `comment` field in the card has been replaced by a popup icon attached to another card. Fields that contain large amounts of data should be 'hidden' behind popup icons. This technique substantially reduces the size of cards and gives a clean professional look.

Load the `startup` program into a text editor.

There is a new variable labelled `chandle` which is initialised to −1 in the `address_card` procedure. This variable stores the handle of the `comment` card. Cards cannot have a handle of −1. S-Base uses −1 to represent a null handle. The `chandle` variable will be checked by the handlers to see if the `comment` card is open.

Load the address card into the format layout editor.



You can see the decrease in size of the card. Fields such as comments tend to contain bulky non-essential data and can be hidden from view. Double-clicking on the comment popup will open another card containing the comment text. The user can leave this card open as long as they wish. Closing the main card should close any popup cards. The popup card should not contain any icons. The popup can be closed with the Close box on the title bar.

Controlling these operations must be done with handlers. Load the click handler labelled `comment click`. This handler manages the `comment popup` icon. The handler first checks if the comment popup icon was clicked on. Next the handler checks to see if the `comment` card is already open. The `chandle` variable will contain the card handle if it has been created. The handler will create the `comment` card if `chandle` is −1. The handler will open the card to the front if it has already been created and place the cursor in it.

A card does not need to be polled if another card has the poll. S-Base checks all open cards if any card is being polled. If no card is being polled S-Base will ignore any open cards. These cards will remain open until a card is polled or the application is rerun.

Check the `add` and `arrows` handler sets. The click handlers for these sets will have the following line added to them:

```
if chandle <> -1 then card update chandle
```

This line will update the `comment` card if it is open. Every time the current `address` record changes, the `comment` card must be updated.

The following line has been added to the `cancel` click handler:

```
if chandle <> -1 then card dispose chandle
```

This line checks if the `comment popup` icon was clicked on and disposes of the comment card if necessary. Any associated card should be closed if the parent card is closed.

Open the format info dialogue for the `comment` card. You will see the `comment` card has a handler set attached to the entire card. Load this handler set and examine the click handler.

The `comment close` handler's purpose is to check if the user has clicked on the Close box on the `comment` card. S-Base will call the handler with **@clickelementlabel** set to `"<close>"`. The handler will dispose of the `comment` card and reset the `chandle` variable to −1. This signifies the card is now closed.

Run your application and popup the card. Leave it open while you find or scroll through addresses. Try closing the `address` card and see if it closes the `comment` card.

22. Menus

There are two menu systems supported by S-Base. You have already looked at program menus and are using them in your application. The second menu system is the standard menu system provided by RISC OS. Creating RISC OS style menus can be quite a complicated task. S-Base provides simple facilities for creating and operating these menus. In this chapter you will add a menu to the Address Book application.

The menu resource

Menus are generally stored in the menu resource. Each menu must have a unique label which is also used to label the resource file. When your application requests that a menu be opened S-Base looks in the menu resource for its definition.

Menus can also be created in programs, procedures and handlers, but by placing them in the menu resource you can find and alter them easily.

When a menu has been created it is assigned a menu handle just like a card handle. The menu and its handle remain valid until the menu is disposed of or the application is terminated. When the user selects an option from a menu it disappears from view, but is not destroyed. It will appear again when required.

Creating menus

Menus are created in a similar way to selectors. A blank menu is created and items are then inserted into it. When you have finished inserting items you can display it on the screen. The operating system only allows one menu to be open on the screen at once.

Once a menu has been created only the contents of each menu item can be altered. The menu structure is fixed and cannot be changed without disposing of the menu first.

Create a new menu labelled `address card` in the menu resource. A text editor will be opened for you to enter a menu definition. The menu you will add will be used to enter cities and countries into the `home_address` fields of the `address` card. Enter the following S code into the menu:

```
local mcities, mcountries, main

| Define the cities submenu
```

```
menu new mcities,"cities","Cities"
menu insert mcities,"Cardiff,Edinburgh,London"

| Define the countries submenu
menu new mcountries,"countries","Countries"
menu insert mcountries,"England,Scotland,Wales"

| Define the address card menu
menu new main,"address card","Address Card"
menu insert main,"Cities,Countries|>Info"
menu set menu main,0,mcities
menu set menu main,1,mcountries

menu set main
```

This menu structure consists of the `address` card menu with two submenus and a card attached. The creation order of each object is very important. Obviously a submenu cannot be attached to the main menu until it has been created and defined. The first step is to create each of the submenus.

The **menu new** command creates a blank menu. The three arguments for this command are a variable in which to store the menu handle, the label of the menu and the title of the menu. The handle of the menu is used in all operations associated with the menu. The handle can be found at any time once it has been created using **@menufind**. The only argument for this function is the menu label. The menu title appears in the title bar of the menu and is restricted to a maximum of 11 characters in length.

The **menu insert** command inserts items into the menu. Each item can be given as separate **menu insert** commands or a combined string separated by commas. The menu string can contain a number of special characters preceding each item:

Character	Meaning
!	Put a tick to the left of this menu item.
~	Fade this menu item so the user cannot select it.
|	Put a line below this menu item. This character must replace the comma if it is used in a combined string.
>	Put an arrow to the right of this entry. The user can move the pointer over the arrow to open a dialogue.

The code above attaches each of the submenus to the address menu. The

menu set menu command attaches a submenu to an item in a parent menu. The three arguments are the handle of the parent menu, the item number to attach the submenu to and the handle of the submenu. The submenus cannot subsequently be altered.

The last command in every menu resource should be **menu set**. This records the menu just created as the 'current menu'. If this command is not present, the menu resource will have no effect and the menu will not be recorded.

When the user selects the `info` item or moves the pointer over its menu arrow, your application will open a card giving helpful information. This will be covered later in this chapter.

Menu maker handlers

When a user clicks Menu over a card or table, S-Base calls the menu maker handler for the element that was clicked on. This should locate the menu using **@menufind** and tell S-Base about it with **menu set**.

As with click handlers, if you want to detect a Menu click which is not over an element of the card, you must attach a handler set to the card's format. Within the handler **@clickelement** will return –1 and **@clickelementlabel** will return an empty string.

It is recommended that you distinguish normal fields from menu fields by either changing the font colour of the field slightly or creating a popup icon next to it. Both the popup icon and the field should be sensitive to a Menu click. You may wish to support a click with Select by creating the menu in the click handler.

The **@menufind** function will return the handle of a menu given its label. If the menu does not exist yet, it will search through the menu resource for a menu with that label and execute it. This menu thus created will be returned by **@menufind**. If S-Base cannot find a menu matching the label you give, then **@menufind** will return –1. This should not normally happen.

Create handler labelled `address menu maker` and attach it to the `address fields` handler set as an S Menu Maker. Enter the following lines of code:

```
local m1, m2, m3
```

```
m3 = @menufind "address card"
m1 = @menufind "cities"
m2 = @menufind "countries"

case @clickelementlabel
when "home_address0","home_address1","home_address2"
   menu clear protect m3,0
   menu clear protect m3,1
   menu clear select m1
   menu clear select m2
   menu set select m1,@trim @elementvalue (ahandle,
@clickelementlabel)
   menu set select m2,@trim @elementvalue (ahandle,
@clickelementlabel)
otherwise
   menu set protect m3,0
   menu set protect m3,1
endcase

menu set m3
```

The first operation of the handler is to find the menu handles of the main menu and submenus. If these menus have not been used before it will search the menu resource and execute them. Next it checks to see which field was clicked on.

Clicking on any field other than the `home_address` fields will protect the city and country items. These should not be selectable as they will be used in the wrong context. The **menu set protect** command sets an item or menu to be protected. The handle of the menu and the item number or text are given as arguments. If an item number is not given the entire menu will be protected.

Clicking on a `home_address` field clears the protection on the city and country fields of the `address card` menu. The **menu clear protect** command does the reverse of the **menu set protect** command and uses the same arguments. Any ticks on the city and country menus are removed with the **menu clear select** command. The arguments are the same as **menu clear protect**.

The **menu set select** command will select an item or menu. The arguments are the same as the previous menu commands. An item number or the text of the item can be given. The handler uses the trimmed value of the `home_address` fields to set the matching menu item. If the field matches one of the menu items it will be ticked. If not the **menu set select** command will do nothing.

The final command in the menu maker handler is **menu set**. This command opens the menu at the location of the mouse pointer.

Menu select handler

When a user selects a menu item S-Base calls the menu select handler. The *menu select handler* must determine which menu item was selected and act appropriately.

Create a handler labelled `address menu select` and attach it to the `address fields` handler set as an S Menu Select handler. Enter the following lines of code into the handler:

```
local m1, m2, ihandle, fieldnum

m1 = @menufind "cities"
m2 = @menufind "countries"

case @menuselect(0)
when 0  {city}
   fieldnum = @val @right(@clickelementlabel,1)
   address.home_address[fieldnum] =
@entrytext(m1,@menuselect(1))
   card update ahandle,@clickelementlabel
   card_altered = TRUE
when 1  {country}
   fieldnum = @val @right(@clickelementlabel,1)
   address.home_address[fieldnum] =
@entrytext(m2,@menuselect(1))
   card update ahandle,@clickelementlabel
   card_altered = TRUE
when 2  {info}
   card new ihandle,"info"
   card show ihandle
   card poll ihandle
   card dispose ihandle
endcase
```

As usual the handler begins by finding the handles of the menus it will reference. Next the handler checks to see which item was selected from the `address` menu. The **@menuselect** function returns the item number that was selected. Because RISC OS only allows one menu to be open at a time the menu handle is not required for this function. The only argument is the level number of the menu hierarchy. The main menu is at level 0. A submenu attached to the main menu is at level 1. A submenu attached to the submenu is at level 2 and so on.

When a city or country item is selected the handler calculates the element

number of the `home_address` array. Next it assigns the city/country name to the `home_address` field and updates the correct field.

The **@entrytext** function returns the text displayed in a specified menu item. The two arguments are the menu handle and the menu item. The menu item was found by using **@menuselect** on level 1 of the menu hierarchy.

When the `Info` item arrow is passed over, S-Base selects the item and calls the menu select handler. The handler creates the info card, shows it and polls it. S-Base will exit the `info` card poll when the menu is closed. The handler finishes by disposing of the `info` card.

Clicking on an item that has a card attached follows the same procedure as above. The handler will not be able to tell whether the item was clicked on or passed over.

Run your application and experiment with the address card menu.

Altering menus

Once set, menus have their structure fixed. The **menu set** command has a variation that allows you to alter the contents of an item. The standard **menu set** command is used to set the menu and open it. Adding a menu item number and a string to the arguments of this command replaces the contents of the menu item with the string.

Altering the structure of a menu is not allowed. You will need to dispose of the menu and create it again. The **menu dispose** command destroys the menu and invalidates its handle. The only argument for this command is the menu handle.

Icon bar menu

S-Base provides your application with the option to create an icon on the icon bar and attach a menu to it.

If a sprite exists in the sprite resource with the same disc label as your application, S-Base will automatically create an icon on the icon bar using this sprite.

If a handler set exists in the handler set resource with the label `iconbar`, S-Base will use this handler set for the icon bar icon. The handler set should have a click handler, menu maker handler and a menu select

handler.

The menu maker handler should open a menu containing an `Info` item at the top and a `Quit` item at the bottom. This is the standard for all good RISC OS applications.

If the 'quit' option is selected the menu select handler should check whether all of the data is saved and quit if it is OK. An application can quit using the **exit** command. This command immediately stops execution and terminates the application.

23. Tables

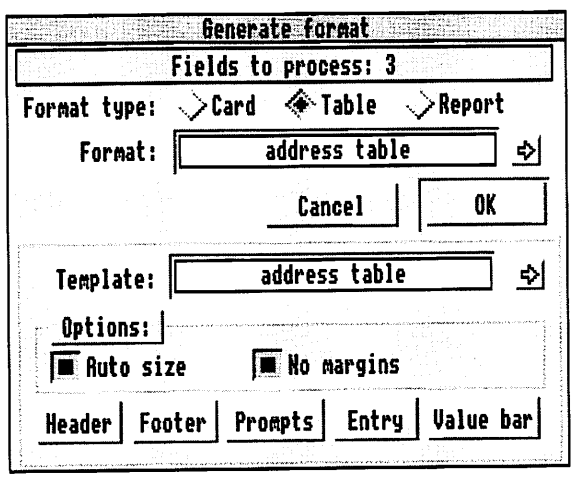
There are two types of data entry device provided by S-Base. You have already had a substantial amount of experience with cards. The other device is the table. Tables are similar to reports in that they consist of a number of entries. However, tables can be used to input data as well as output it. In this chapter you will create a table to edit your address database and study its operation.

Creating tables

The first step in creating a table is to create a format and some handlers. A request entry handler is provided when S-Base wants to generate an entry. Each entry in the table is treated like a card and has all the usual handlers to support it.

There are two new handlers to control the movement between entries. The enter entry and exit entry handlers give you a chance to set up a new entry and save an old entry.

Create a blank handler set labelled `address table`. Select the `name`, `home_address` and `birth_date` fields from the `address` file and start AFG for a table format labelled `address table`.



The AFG dialogue for tables is similar to the AFG card dialogue. The top half of the dialogue contains the standard format information. The bottom half contains a template field, an auto size option, an option for generating

the table without margins and five setting icons.

Use the `address table` template for the table's template and ensure that 'Auto size' and 'No margins' are on.

The 'header' setting allows you to define the font style for the header. This is created at the top of the format. This should be turned off for this table.

The 'footer' setting allows you to define the font style for the footer. A single text prompt will be created in the column footer area. You will need to add any additional footer elements manually. This should be turned off for this table.

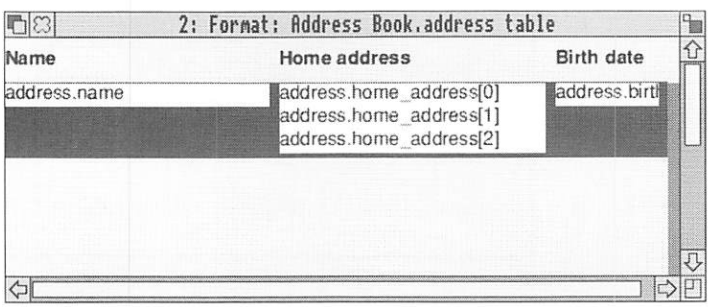
The 'prompt' setting allows you to define the font style for the field prompts. These are positioned directly above the entry as part of the header. Set this on.

A common practice is to enclose the headers, prompts and footers with filled rectangles. This makes them stand out from the body of the table and emphasizes their fixed nature. The ordering of the elements will decide whether the rectangle is displayed on top or behind the prompts. Obviously this is important if the rectangle is filled.

The 'entry' setting allows you to define the handler set for the entry area and font style for each of the entry items. Set this on.

The 'value bar' setting allows you to define the font style for the value bar. Set this off.

Click OK to generate the format. S-Base will display the new table format in a format editor.



You will need to alter the size of the `birth date` field and add formatting

information (`:/m`). Attach the `address table` handler set to the format using the format info dialogue. The click handler will be used to close the table when the Close box is clicked.

The rules for reports with regard to the ordering and enclosing of objects also apply to tables. All format elements must be preceded by an area element. The elements associated with an area must lie between that area and the next area on the format element list. The entry area should be last in the format, followed by the entry elements.

There are five areas in a table. The 'column header' area and its prompts perform the same function as those used in reports. The column header in the table is fixed in position and does not scroll.

The 'column' area defines the size of the scrolling part of the table. All tables have a vertical scroll bar but there is no horizontal scroll bar.

The 'entry' area and its fields perform the same function as those used in reports. The entry area can contain prompts, fields, sprites, drawings, rectangles and lines. The entry area cannot contain selectors or icons. Multi-line fields are supported in entry areas.

If you require an icon to be used in the entry area it is recommended that you use a sprite or drawing element. Use the mouse click and menu maker handlers to trap any mouse events.

There is no page area for tables. The size of the page is determined by the dimensions of the template.

A handler set must be attached to the entry area so that it can control operations affecting the entry as a whole. Request entry, enter entry and exit entry are all examples of these operations.

Each element inside the entry area should be treated as if it was part of a card. Handler sets should be attached to any element that would require one for normal card operations.

The 'column footer' area and its prompts perform the same function as the 'column header' area. The column footer in the table is fixed in position and does not scroll.

The final area is the 'special' area. This contains the value bar, which is attached to the bottom of the window and performs the same function as a value bar attached to a card.

Most tables used for simple data entry will only use two of the handlers available. The request entry handler is used to redraw the entry. The exit entry handler is used to save the changes to disc. These handlers will now be explained further.

Request entry handler

The request entry handler performs the same function in tables as it does in reports. Its purpose is to supply information to S-Base when it needs to generate an entry. In reports the request entry handler was called in sequence for each entry down the page. In tables the request entry handler can be called for any entry in the table at any time. The request entry handler is used in the following situations:

1. For redrawing the visible area of the table when it is created. After the table has been created and opened S-Base will call the request entry handler for each entry visible on the screen. S-Base caches this information and will not need to load it again unless the cache needs refreshing.

2. Redrawing table entries when it is scrolled. Scrolling the table up or down will require S-Base to redraw the new entry displayed. If the entry is not cached S-Base will call the handler for entry information.

3. The cursor entering an entry. If the entry is not cached S-Base will call the handler for entry information. Usually this is a result of clicking into an entry.

It is briefly worth mentioning how S-Base handles scrolling and the request entry handler when the cursor is in an entry. Before calling the request entry handler S-Base temporarily stores all the data for the current entry. If it did not do this the data for the current entry would be overwritten by the request entry handler. After returning from the handler S-Base restores the data for the current entry.

Create a new handler labelled `address table request` and attach it to the `address table` handler set as an S Request Entry handler. Enter the following line into the handler:

```
load address,@entry+1
```

This line will load the `address` record S-Base needs to generate the table entry.

Exit entry handler

The exit entry handler is called whenever the cursor leaves an entry. This can be the result of pressing the up or down arrows or clicking out of an entry. The main purpose for this handler is to save any changes in the entry back to disc.

Create a new exit entry handler labelled `address table exit` and attach it to the `address table` handler set. Enter the following line into the handler:

```
if @entryaltered then update address,@entry+1
```

This line will check if any of the data in the entry has been altered. If so, it will update the `address` record on disc.

There is one last object you will need to add to your table. In order to close your table the user will need to click on the Close box on the title bar.

Create a new S click handler labelled `address table click` and attach it to the `address table` handler set. Enter the following line into the handler:

```
if @clickelementlabel = "<close>" then table exit poll
thandle
```

This line will exit the table from the poll and return control to the `startup` program when the user closes the table.

Using tables

Your table is now complete so it is time to use it. Load your `startup` program into a text editor and insert the following line into the program's menu definition:

```
pmenu insert mhandle,"","3",Address table"
```

You will need to renumber the key numbers in the other pmenu items. Add the following line to the case statement for the menu:

```
when 2
   proc address_table
```

You will need to renumber the **when** commands below this code.

Add the following procedure to the top of the program:

```
| Create and display the address table

def proc address_table
   local thandle
   table new thandle, "address table", @recordcount
address
   table show thandle
   table poll thandle
   table dispose thandle
enddef
```

Run your application and test the `address table` option.

You will be able to scroll up and down the table using the up arrow, the down arrow, page up, page down or the scroll bar. If you alter any addresses, your application will save the changes.

Attaching tables to cards

A useful extension to tables would be to have a card associated with the table that can contain icons and fields pertaining to the operation of the table. For example, the card could be a tool bar for the table containing `add` and `quit` icons. This example has been implemented in the !Address5 tutorial application.

Load this from the tutorial directory on your S-Base work disc. Load the `address combo` format into the format layout editor.

This format contains standard `add` and `quit` icons. Below them is an element labelled `table`. This is a 'special' area element and will be used to attach the `address table` table to the card. When the table is attached to the card it will be positioned over the `table` element. Rather than the table's 'column' area, the bounding box of the `table` element will determine the size of the table displayed.

When a table is used in this situation it can be considered as a sophisticated selector. It will function as a normal table though it will be embedded into the card. S-Base will only use the column area, entry area and entry elements from the table. For this reason, the `address table` template is no longer used and has been removed. The *card* will need to display prompts for the column headers and footers. These can be created with AFG and dragged into the card.

The `address table` template and format will no longer contain a Close box. The Close box has been replaced with the `Quit` icon in the `address`

combo card. The address combo handler set is now attached to both the card and the table. The address table handler set no longer exists. The address combo handler set has also been attached to the Quit icon. The click handler attached to this handler set will exit from the **card poll** when the Quit icon is clicked.

Load the startup program into a text editor and examine the address_table procedure.

The procedure first creates the address table then creates the address combo card. The next step is to attach the table to the card.

The **card set table** command attaches a table to an element in a card. The element in the card must be a format area element in the 'special' area. The three arguments are the card handle, the element label and the table handle. The table must be created before it is attached to the card. You must dispose of the card before you dispose of the table.

All events referencing a table attached to a card will still treat the table as a separate entity and continue to use any handler sets attached to the table.

Run the application and experiment with the table. Use the Quit icon to close the card and table.

Protected Fields

Protecting fields in tables is similar to cards. The **table set protect** command is used to protect individual fields in any entry. The arguments for the command are the table handle, the entry number and the element label.

The **table clear protect** command reverses the action of **table set protect**. The arguments are the same for both commands.

Table extension

Tables are useful for editing and viewing records but run into problems when it comes to adding records. The address table table is fixed in length and there are no empty entries to add new records. For this reason S-Base allows a table to be extended or contracted at will. The S extension handler is called whenever a table attempts to extend.

There are three situations where a table will attempt to extend. The first occurs when a table containing no entries is opened. S-Base will attempt to

extend the table so that the table contains something.

The second situation occurs when the user tries to scroll below the bottom of the table. If the user presses the down arrow when at the bottom of the table S-Base will attempt to extend the table by one entry. If the user pressed Page down when at the bottom of the table S-Base will attempt to extend the table by a page of entries.

The third situation occurs when the application requests S-Base to extend the table. An application can 'extend' the table by a positive or negative amount. The **table ext** command will extend the number of entries in a table by a specified amount. The two arguments are the table handle and the number of elements to extend by. A negative number will contract the table.

Alter the `combo click` handler to match the following code:

```
case @clickelementlabel
when "add"
   if @trim address.name <> ""
      ext_ok = TRUE
      table ext thandle,1
      table set handle thandle
   else
      error 1,"Please enter a valid name."
   endif
when "cancel"
   card exit poll chandle
endcase
```

Clicking on the `add` icon will check if the user has entered a name for the current record yet. If they have it will set `ext_ok` to `TRUE` and extend the table by one record. The variable `ext_ok` is used to control when the table can be extended. If the name has not been entered the handler will prompt them to do so. Your application should always encourage the user to complete the current entry before adding another.

Table extension handler

The table extension handler has three purposes. The first is to check that it is OK to extend the table. The handler must return true (1) if it is OK and false (0) otherwise.

The second purpose is to check that the number of entries to extend by is acceptable. The **table set ext** command will set a new extent for the table. The two arguments are the table handle and the new table size. S-Base may

request that the table be extended by 50 entries. Your handler will then have to decide how much extension to actually allow.

The third purpose is to ensure there is data available to extend with. If the table is displaying records it may be necessary to create a new record so that the new entry will have something to display.

Create an S extension handler labelled `address table extend` and attach it to the `address combo` handler set. Enter the following lines of code into it:

```
local ec = @entrycount @handle

table set ext @handle,@min(@ext @handle,ec+1)
if ext_ok && (@trim address.name <> "" || !ec)
   zero address
   create address
   ext_ok = FALSE
   table set entry @handle, @ext @handle
   table set element @handle, "name"
   return TRUE
else
   return FALSE
endif
```

The first operation of the handler above is to find how many entries are in the table. The function **@entrycount** returns the current number of entries in the table. The only argument is the table handle.

You should note the use of the **@handle** function. This function returns the handle of the current object, in this case the table handle. You should never use a table handle variable in an extension handler. This is because if your table starts for with no entries (i.e. 0 is passed as the entry count parameter to **table new**), S-Base will immediately call the extension handler to create an entry. At this stage the **table new** command will not have finished, so will not have written the handle of the table into the table handle variable. `thandle` would not yet contain the table's handle and would produce an error if used.

The next line of code sets the new extent of the table. The **@min** function finds the minimum value of all its arguments, the first of which is the amount the table is attempting to extend by. The function **@ext** returns the number of entries the table would like to extend to. The second argument to **@min** is the current extent plus one. The reason for finding the minimum of these to values is to ensure the table can only extend by one

entry at a time. Your application may ask the handler to extend by a negative amount in the future, to allow the user to delete records.

The next check is to determine if it is OK to extend the table. It is OK to extend if `ext_ok` is `TRUE` and the current record has a name entered or there are no records in the table. If it is not OK the handler will return `FALSE` and S-Base will not extend the table. If it is OK to extend the table the handler will create a new blank record, position the cursor in the new entry and return `TRUE`. S-Base will then extend the table and load your new record for editing.

Run your application and try out the table.

Table insertion and deletion

Sometimes it is necessary to insert or delete an entry from the middle of the table. S-Base provides two commands for handling these situations. The **table insert** command inserts a number of entries at a specified location in the table. The three arguments for the command are the table handle, the entry number to insert before and the number of entries to insert.

The **table insert** command will only insert entries into the table and shuffle the screen down if necessary. It does not affect the data used by the request entry handler. For example, the command does not insert blank entries into the `address` file. You must do this yourself after the **table insert**.

The **table insert** command does not extend the table. You will need to do this before inserting entries otherwise the entries at the bottom of the table will be lost. The following code will correctly insert a single record into a table of records before the current entry. The records are loaded by the request entry handler using the array `recnums`:

```
table ext thandle,1
table insert thandle,@entry,1
array insert recnums[@entry],1
```

The **array insert** command insert elements into an array. The two arguments are the array variable with a start element and the number of elements to insert. The **array insert** command does not increase the number of elements in an array, so elements at the end of the array will be overwritten.

The **table delete** command performs the reverse operation to the **table**

insert command. It deletes entries without contracting the table. The following code will correctly remove the current entry from a table. The records are loaded by the request entry handler using the array `recnums`:

```
table delete thandle,@entry,1
table ext thandle,-1
array delete recnums[@entry],1
```

The **array delete** command deletes elements from an array. The two arguments are the array variable with a start element and the number of elements to delete. The **array delete** command does not decrease the number of elements in an array.

24. Advanced files

In previous chapters you have created files, attached indexes, and searched for records. These operations are the basic operations of a database. This chapter will describe some of the more advanced aspects of files. Work through the examples at your own pace and ensure you understand each new concept before moving to the next.

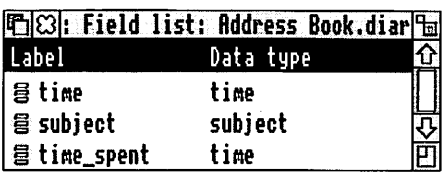
Floating structures

A floating structure is a special type of file that is used for describing complex data types. Floating structures are used to group file fields together into a complex data type. Using floating structures in this manner will allow you to define a file field made up of other file fields. The most common use of floating structures is to create file fields that are arrays of structures.

The example in this chapter is a record of appointments you have had with people in your address book.

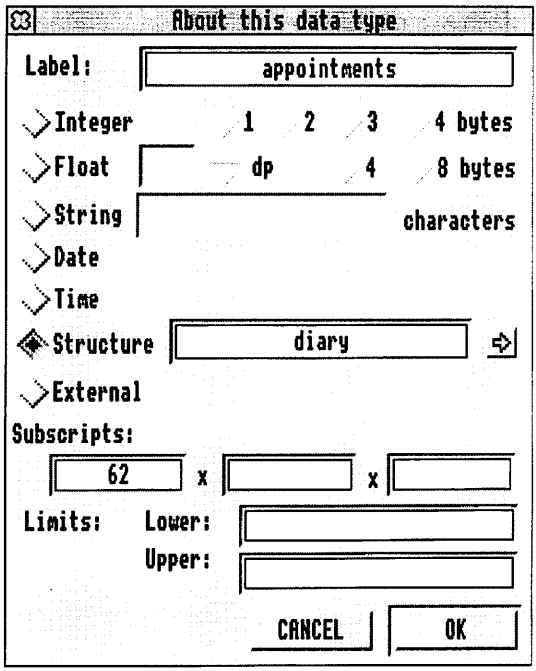
Open the file list viewer and create a new file labelled `diary`. Set the file type to floating structure.

You will notice there is no `system` field in the file. This is only used in disc files. Create a field labelled `time` and a data type labelled `time`, of type time.



Next create a field labelled `subject`. You will need to create a data type labelled `subject` that is a string of 30 characters. Finally create a field labelled `time_spent` using the data type `time`.

Create a new data type labelled `appointments`. Click on the structure popup and select `diary`. This data type now consists of the fields `time`, `subject` and `time_spent`.



An appointments diary with space for only a single day of information is not very useful. Enter `62` into the first subscript field. Your data type now consists of `62` sets of the fields `time`, `subject` and `time_spent`. This will be enough to hold at least two months worth of appointment data.

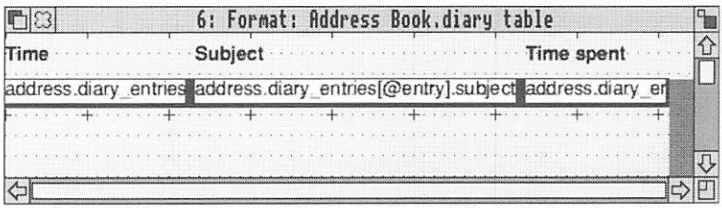
Add a new field to the `address` file labelled `num_entries`. Create a data type labelled `integer1` and use it in the `num_entries` field. It should have a type of 1 byte integer. This field will be used to store the number of diary entries you are going to use in your `address` record.

Create a field labelled `diary_entries` in the `address` file using the `appointments` data type.

To reference a structure field you must add the structure field to the end of the file field name. These must be separated by dots. For example, the field you have just created can be referenced by `address.diary_entries[0].time`. This is read as the `time` field in the first set of `address.diary_entries` of the `address` file.

Create a blank handler set labelled `diary table`. Select the `diary` file

and create a table format using AFG. Label the format `diary table`. Use the `diary table` template and handler set. The 'prompts', 'entry' and 'no margin' settings should be on. You will need to horizontally size down the `time_spent` field, and entry and column areas to fit inside the window.



Once the table has been created you will need to replace the `diary` file name with `address.diary_entries[@entry]` for each of the field expressions. Append `:/t` formatting information to the `time` field expression. Append `:/i` formatting information to the `time_spent` field expression.

You will not be able to use the `diary_entries` field of the `address` file in AFG. AFG does not support structure fields so you must create the table by hand or use the original structure file and alter it.

The `diary table` table will be opened as a scared object so you will need to attach a handler set to close it. Attach the `diary table` handler set to the `diary table` format. Create a click handler labelled `diary table click` and attach it to the handler set. Enter the following line of code into the handler:

```
if @clickelementlabel = "<scared>" then table exit poll
thandle
```

The handler will check if the user has clicked outside the table and, if so, exit the poll. Whenever the user clicks outside a card or table which is being polled, S-Base will call the format handler set with the label `<scared>`. In theory all cards and tables are scared. The reason most cards and tables do not close whenever the user clicks outside them is that the format handler sets do not check for the `<scared>` string.

The **close** option used with the **card poll** command simply allows the application to substitute the `<scared>` label with another label S-Base will use. Because tables cannot have elements outside the entry area there is no **close** option for the **table poll** command.

You will need an icon on the address card to open the diary table. The debug icon has become redundant so replace it with a diary icon. Rename the debug handler set to diary. Rename the debug click handler to diary click and replace the handler's contents with the following:

```
local thandle

table new thandle,"diary table",address.num_entries
table show thandle
table poll thandle
table dispose thandle
```

When the diary icon is clicked on the handler will open the diary table with the number of entries specified by address.num_entries.

Load the address template into the template layout editor. Rename the debug icon to diary. Change the text override from Debug to Diary. Load the address format into the format layout editor. Rename the debug icon element to diary.

Create an S extension handler labelled diary table extend and attach it to the diary table handler set. Enter the following lines of code into the handler:

```
local ec = @entrycount @handle

table set ext @handle, ec + 1
if !ec
   address.num_entries += 1
   return TRUE
endif
use address.diary_entries[@entry]
if diary.time <> @timest "" && @trim diary.subject <> ""
&& diary.time_spent <> @timesi ""
   address.num_entries += 1
   return TRUE
else
   return FALSE
endif
```

The handler first finds the number of entries currently in the table then increases the extent of the table by one. Following this it checks the number of entries in the table. If there are no entries in the table it will increment address.num_entries by one, return TRUE, and exit from the handler. This is essential because a table of zero size will have an **@entry** value of –1. This will cause an 'out of range' error for the arrays.

The next instruction executed will set up the **use** command. The **use** command's argument will map a floating structure onto a file field. This mapping will point to the start of the structure in the file field. The application can now use the label of the floating structure in code, instead of the full file and field names, and S-Base will understand which field is being referenced.

For example, one of the sets of `diary` entries is mapped to `address.diary_entries[@entry]`. If `@entry` was `10` when **use** mapped the structure, `diary.time` would reference `address.diary_entries[10].time`. If the value of **@entry** was to change you would need to set up **use** again. The **use** command is a handy short cut when you are using long sophisticated data structures.

After setting up the **use** command the handler checks each of the fields in the current entry to make sure they have been completed. If this is true the handler will increment `address.num_entries`, return `TRUE` and exit the handler.

Create an exit entry handler labelled `diary table exit`. Attach this handler to the `diary table` handler set and enter the following line of code:

```
if @entryaltered then card_altered = TRUE
```

The exit entry handler checks to see if the user has altered the entry and sets `card_altered` accordingly. There is no need to save any data in this exit handler.

To complete the alterations you will need to change all the occurrences of the icon label `debug` to `diary`. These can be found in the `startup` program, `save click` handler and `add click` handler.

You have done very well to reach this stage. It is time for you to run your application. Try out all the features and examine how each part interacts with the other.

Variable length records

Open the file list viewer in your application.

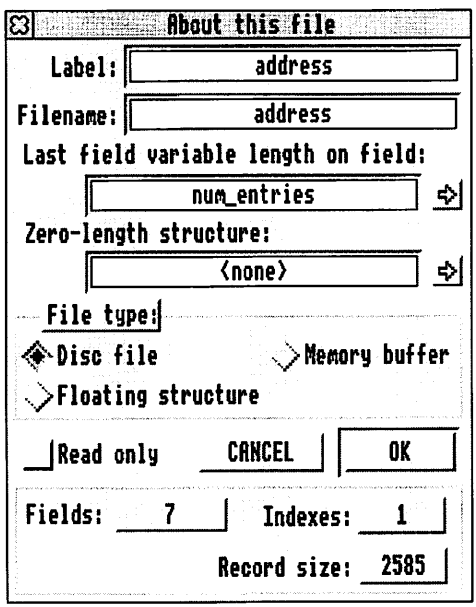
You can see the record size of the `address` file is quite large. A number of `address` records would occupy a considerable amount of disc space. Yet the majority of each record is empty at this stage. It would be more

efficient if each record only stored the data that is used and grew or contracted as it needed to. A variable length record is just such a record. It will allow you to expand its size to a specified maximum length or contract it to a specified minimum length. If a record expands or contracts in the middle of the file S-Base will automatically shuffle all the other records on disc.

Ctrl-double-click on the `address` file to open the file info dialogue.

Examine the two fields below the filename. These fields are used to control the variable length aspects of a file. S-Base allows only the last field in a file to be variable length. This field must be an array of any type with the number of elements in the array being variable.

The first of the two fields sets the variable length counter. Click on the menu popup and select `num_entries`. S-Base will use this field as a counter to store the number of elements in the array to save to disc. For example, if `address.num_entries` was equal to `5` S-Base would only load or save `5` elements of the `address.diary_entries` array. The value of `address.num_entries` can only contain `0..62`. This is the range of the array you have defined.



Leave the second field set to <none> and click 'OK'. This field will be explained in the next section. Check the free space on your disc drive. Try running your application. S-Base will detect the file structure has changed to variable length and will import only the valid data in your address file. Now check the free space on your disc drive. It will have increased as each record will only be storing the number of diary entries specified in address.num_entries.

Zero length structures

The second field in the file info dialogue is the zero length structure field. If a variable length record contains no variable length elements S-Base will replace the current record structure with the zero length structure. Both structures must match byte for byte except the variable length component. From this field onwards the zero length structure can contain any type or number of fields. For example, an invoice may have a number of entries whereas a receipt has just one entry though needs extra data stored in a different format.

Listed below are two file structures. The trans file is used to store transactions on disc and is variable length on the num_trans field. The items array contains a structure of numeric fields. This file structure is good for storing invoices of many items though does not have the appropriate fields to store receipt information.

The receipt file is a memory file specially designed to store receipt information.

field #	**trans**	**receipt** (zero length struct)
1	system	system
2	customer	customer
3	date	date
4	trans_type	trans_type
5	num_trans	num_trans
6	items[0..??] (structures)	value
7		payment_type
8		cheque_number

When the value of num_trans is greater than zero (an invoice) S-Base will load or save a record using the trans file structure. When the value of num_trans is equal to zero (a receipt) S-Base will load or save a record using the receipt file structure. Both file structures are stored in the one

disc file.

The first 5 fields are identical so both structures will contain the same common data. You could use `trans.customer` or `receipt.customer` in your code and receive the same value. Either file label can be used on the common data fields.

The fields from 6 onwards are different. For field 6 the following would be true. If the record was an invoice (> 0 items) you could only use `trans.items[n]...` in your code. If you were to use `receipt.value` in your code you may receive an error because the `receipt` structure would not match the data on disc.

If the record was a receipt (0 items) you could only use `receipt.value` in your code. If you were to use `trans.items[n]...` in your code you may receive an error because the `trans` structure would not match the data on disc.

Select

Load the Address6 application from the tutorial directory of your S-Base working discs.

S-Base provides a high speed searching command called **select**. The **select** command rapidly searches an entire database file for matching records. If S-Base finds a match it will call a user-defined procedure with the number of the record it has found. Your procedure definition must include a single parameter. S-Base will pass the record number to your procedure in this parameter.

Load the `startup` program into a text editor. Add the following procedure to the top of the program:

```
| Procedure for select command

def proc found_record(rec)
   load address,rec
   print address.name
   return FALSE
enddef
```

The procedure has a parameter labelled `rec`. This variable will contain the number of the record S-Base has found. This procedure will load the matching `address` record and print it to the command line. More often this procedure would store the record numbers in an array. These records

would be processed or printed in a report.

The procedure must always return a true (1) or false (0) value. This informs S-Base whether to terminate the search. True will terminate the search; false will continue with the search until S-Base has checked each record in the file. For example, if you only wanted to find 10 matches you would use a counter and return `TRUE` when it reached 10.

The **select** command has three mandatory arguments followed by the search data. The first argument is the file to search. The second argument specifies the index to search and the order in which it will be searched. The third argument is the procedure to call when it finds a match. Following these arguments is the search data separated by commas. This is the same as for the **@find** function. If no search data is provided the **select** command will match all records in the file.

In the example below, S-Base will search the `name` index of the `address` file for the string `"steve"`. If it finds a match it will call the procedure `found_record`.

Insert the following procedure below `found_record`:

```
| Search for some data

def proc search_address
   select address,"name", "found_record", "steve"
enddef
```

Add an item to the main menu to call this procedure and run your application. You may need to create some records that contain `steve` in the name.

Filters

The **select** command also provides a filter option. A filter is an expression string that will be evaluated against the contents of each record. If the **select** command finds a match with the search data it loads the record and evaluates it with the filter. If the result is true (non-zero) the record number is passed to the procedure.

The only disadvantage with using a filter is the requirement for S-Base to load each record off disc. Searching without a filter will only load the index off disc; this is considerably smaller. However, in some circumstances the filter does not require the record to be loaded. Consult

the S Reference for a description of these cases.

Replace the `search_address` procedure with the following:

```
| Search for some data

def proc search_address
   select address,-1,"found_record" for "@trim @toupper
address.name >= ""STEVE"""
enddef
```

In this example `-1` is used for the index number. Index `-1` is the physical order the records are stored on disc (record number order). If this index is used, search data cannot be provided.

If a filter is provided with the **select** command you should use index `-1`. This index will give a considerable increase in searching speed for linear searches (having to search all records) because S-Base can load large blocks of adjacent records from disc.

The **for** command is used to separate the search from the filter. The filter in the example above will compare the trimmed upper case form of `address.name` with the string `"STEVE"`. To use quotes within a string you will need to use `""`. S-Base will interpret double quotes in a string to mean the quote character (ASCII 34). See the S Reference (**select**) for more information about filters.

Run your application and compare your results with the first **select** command you used.

File commands and functions

This section will briefly list the file commands and functions that are not covered in tutorial examples. To see a detailed description of these operations and their arguments consult the S Reference.

delete marks a record on disc as deleted and removes it from all indexes. The record is still intact on disc with only the `system` field being altered.

flush flushes the contents of the I/O buffers to disc.

pack removes all deleted records from disc and compacts the file. Note: **pack** will change the record number of each record in the file.

save saves a record to disc. Note: **save** does not update the

	indexes.
reindex	rebuilds and sorts an index. Useful for constructing an index after importing data from another source.
zap	initialises a database file and its indexes. This command will destroy all data in the file.
@bof	finds the first record in an index.
@deleted	returns true (1) if a record is deleted.
@indexcount	returns the number of indexes in a file.
@loadindex	loads the record numbers in an index into an array.
@order	orders an array of records based on another array of records. This is mainly used to order an array of records to the same order as an array containing an index.
@recsize	returns the size of a record in bytes.
@tof	finds the last record in an index.
@vrecsize	return the size of a variable length record in bytes.

Progress functions

Progress functions are used to inform your application of how far S-Base is through a time-consuming operation. There is only one operation in S-Base that uses progress functions.

The **reindex** command uses a progress function to notify the application of S-Base's progress when it is reindexing a large file.

Enter the following code into the `address` procedure library:

```
def proc rprogress (pc)
   upto = todo * pc // 100
   card update chandle,"upto"
enddef

def proc reindex_file
   local upto, todo, chandle
   upto = 0
   todo = @recordcount address
   card new chandle,"reindex"
   card show chandle
   card poll until 0, chandle
   reindex address, 0, rprogress
   card dispose chandle
```

```
enddef
```

The `reindex_file` procedure will reindex index `0` of the `address` file. First the procedure sets up two variables. `upto` stores the current record number the **reindex** command is up to. `todo` stores the total number of records in the file. A card using the format `reindex` is created, shown and polled. The card displays the value of `upto`.

The **card poll until** command will poll a card until a null event (0) is received from the Window Manager. A null event will only be received after the card has been redrawn on the screen.

The **reindex** command has an optional third argument. This argument is the name of a procedure that is called when S-Base feels it is necessary (usually this is about every half a second). This procedure is similar to that in the **select** command. The procedure must have a single parameter which will be passed the percentage S-Base is through the current operation. This value is converted into a record number using `todo` and `upto`. The card is updated and the user will see which record number the **reindex** command is up to. The **card update** command does not require S-Base to be polling the Window Manager to update the card on the screen.

Add an item to the main menu which calls `reindex_file` and run your application. Because of the number of items in the menu it would be a good idea to change the menu to two columns.

If you only have a small number of records in the `address` file the **reindex** command may never get a chance to call the progress function.

Importing and Exporting

S provides a number of commands and functions for importing and exporting data off the database.

@openin opens a file to read data from. The function resets the file pointer to the beginning of the file and returns the handle of the file. The only argument is a string containing the file's pathname.

@openout creates a file to write data to. The function resets the file pointer to the beginning of the file and returns the handle of the file. The only argument is a string containing the file's pathname. If the file already exists it will be destroyed.

@openup	opens a file for reading and writing data to. The function resets the file pointer to the beginning of the file and returns the handle of the file. The only argument is a string containing the file's pathname. If the file already exists it will not be destroyed.
close	closes a file. The only argument is the handle of the file.
@bget	reads a byte from an open file. The only argument is the handle of the file. The file pointer of the file is incremented by one.
bput	writes a byte to an open file. The first argument is the handle of the file. A list of values for writing to the file follow next. The file pointer of the file is incremented by the number of bytes written.
import	reads binary values from an open file. The first argument is the handle of the file. A list of variables for storing data from the file follow next. The file pointer of the file is incremented by the number of bytes read.
archive	writes binary values to an open file. The first argument is the handle of the file. A list of values for writing to the file follow next. The data is written in binary file format. No type information is written to the file. The file pointer of the file is incremented by the number of bytes written.
print to	writes text to an open file. The first argument contains the handle of the file. The remainder of the command is a normal **print** command. The output from the **print** command will go to the file. This facility is primarily for supporting the CSV file format.
@ptr	returns the file pointer of an open file. The only argument is the handle of the file.
set ptr	changes the value of the file pointer for an open file. The first argument is the handle of the file. The second argument is the new file pointer. The file pointer of a file is the offset into the file that data will be read or written to.

@ext	returns the size of an open file. The only argument is the handle of the file.

25. External data

In the previous chapter you have added an appointment diary to your address book application. This diary used S-Base's variable length record facilities. S-Base also provides another system for storing unknown amounts of data in a record. The *external* data type is a special type that allows a file field to store a variable amount of data. There is no limit to the number of external data fields in a file.

External data types are an extension of the way S-Base handles an application's resources. S-Base stores an application's procedures, menus and handlers together in one large file called a *disc heap*. The disc heap is managed by S-Base and is completely 'transparent' to the application and user. The internal operations of a disc heap are beyond the scope of this tutorial so you should just leave it to S-Base.

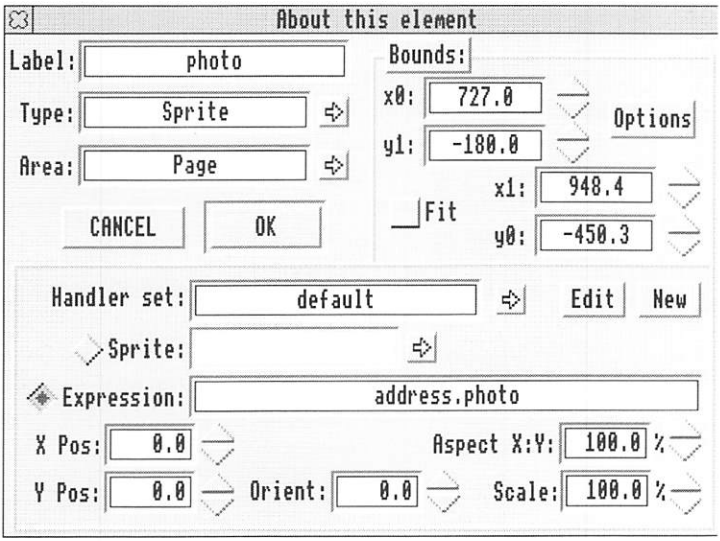
Storing data in this fashion has many advantages. With only one large file stored on disc S-Base is not limited by filing system constraints such as the number of files per directory etc. The disc heap can expand and contract with ease and does not need to be loaded into the computer's memory to function. The disc heap also retrieves large amounts of data very quickly.

A file field of external data (external field) does not actually contain the data you are storing but the handle of a data block in the disc heap. S-Base uses this handle to save and load data blocks from the heap. The value of the handle must **never** be altered by the application. If it is changed by the application all data in the disc heap could be lost. S-Base will adjust the handle's value as it needs to.

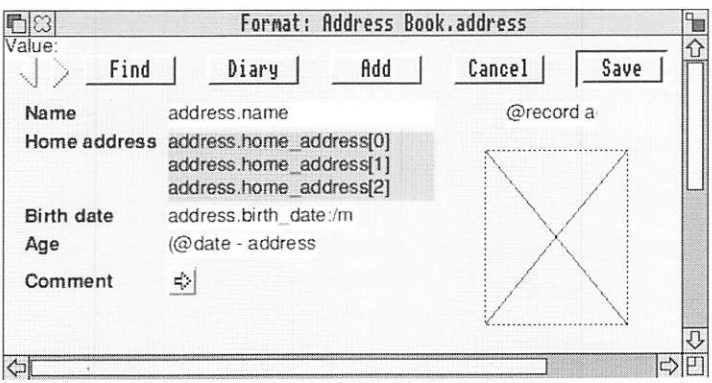
Create a field in the `address` file labelled `photo`. You will need to create and use a data type labelled `extern`, which will be of type external. The `photo` file field will point to a sprite stored in the disc heap. Load the `address` format into the format layout editor and create a sprite labelled `photo`.

Click on the expression button and enter `address.photo`.

The expression field for sprite and drawing elements is used to enter a file field containing an external handle.



S-Base will recognise this element as an external sprite and automatically use the handle in `address.photo` to display a sprite stored in the disc heap. How this process works is beyond the scope of this tutorial.



S-Base will not display the sprite in the format layout editor. Position the sprite at the right side of the format. Add the following line to the `left arrow and right arrow` **whens** of the `arrows click` handler:

```
card update ahandle,"photo"
```

This will update the `photo` element for each record.

Importing external data

Before you can use the photograph sprites, the data must be imported from an external source. The S-Base tutorial directory will contain a sprite file labelled `photos`.

Insert the following procedure at the beginning of the `startup` program:

```
def proc import_photos
   local fd, b, rec
   buffer new b
   for rec = 1 to 5
      fd = @openin("<Photos$Dir>."+@str (rec))
      set ptr fd,12
      buffer load ext b,fd
      load address,rec
      buffer save b,address.photo
      save address,rec
      close fd
   next
enddef
```

This procedure will import five sprites from separate files in the `!Photos` directory. The files are labelled '1' to '5' and will be inserted into the records with the same number.

The first command encountered by S-Base is the **buffer new** command. This command creates a memory buffer and returns the buffer handle. The buffer size will be 0 bytes initially.

Inside a **for..next** structure each of the photograph files is opened with **@openin** and the file pointer is set with **set ptr** to an offset of 12 bytes. This is to avoid the sprite area header attached to the beginning of each sprite file. For more information on the format of sprite files you should consult the RISC OS Programmers' Reference Manual.

The **buffer load ext** command loads the sprite file into the buffer. The two arguments for this command are the buffer and file handles. The buffer will automatically extend or contract in size to match the file being loaded.

The **buffer load ext** command is a variant of the **buffer load** command. The **buffer load** command is functionally identical to **buffer load ext** except it does not automatically adjust the buffer size. To adjust the size of a buffer, use the **buffer set ext** command. The two arguments for this command are the buffer handle and the size in bytes to set the buffer.

After loading the sprite into the buffer the procedure loads the `address` record from disc. The **buffer save** command saves the contents of the buffer to the disc heap and stores the heap block handle in `address.photo`. The two arguments for the command are the buffer handle and an external field.

The procedure finishes by saving the `address` record to disc and closing the external sprite file.

Insert the `import_photos` procedure into the main menu and run your application. Insert the S-Base working disc into floppy drive 0 and import the photographs. Next try viewing an `address` record. The records numbered 1..5 will display photographs with the remainder displaying a blank field. External fields that do not contain external data will not be displayed by S-Base.

To find the size of an external field use **@ext**. The argument for this command should be the external field. S-Base will extract the field's block handle and use it as the file handle. For example, `@ext address.photo` will return the size of the photograph stored in the current `address` record.

To find the file pointer of an external field in the disc heap use **@ptr**. The argument for this command should be the external field. S-Base will extract the field's block handle and use it as the file handle.

The greatest use for external fields is for storing very large data objects like *Acorn Replay* movie files. These can contain many megabytes of sound and graphics. It would not be possible to load these into a buffer so S provides a variation on the **import** and **export** commands. These commands also have an **ext** option that will allow you to specify the amount of data to be copied. This should be specified immediately after the field you are using. For example,

```
import fd,address.photo ext 300
```

will import the first `300` bytes of the photograph file into the `address.photo` field and increment the file pointer by `300`. This data will be transferred directly from the source file into the disc heap and will not be loaded into memory. If the number of bytes to copy is not provided, S-Base will copy the entire file starting from the current file position.

26. Graphics

S provides a number of graphics commands. This chapter will examine these and illustrate how they can be used in the multi-tasking window environment.

Graphics commands

This section briefly describes the graphics functions provided by S-Base. To see a detailed description of their operation and arguments consult the S Reference.

circle	draws a circle on the screen. There is a fill option for drawing filled circles.
clg	clears the current graphics window.
draw	draws a line on the screen.
gcol	sets the current graphics colour.
move	moves the graphics cursor.
origin	sets the graphics origin of the screen. All graphics coordinates are given relative to the origin.
plot	is a general purpose graphics command. It draws points, lines, circles, rectangles, triangles, etc.
point	sets the colour of a pixel on the screen.
rectangle	draws a rectangle on the screen. There is a fill option for drawing filled rectangles.
vdu	writes a value to the VDU drivers.
@point	returns the colour of a pixel on the screen. If the coordinates are off the screen −1 is returned.

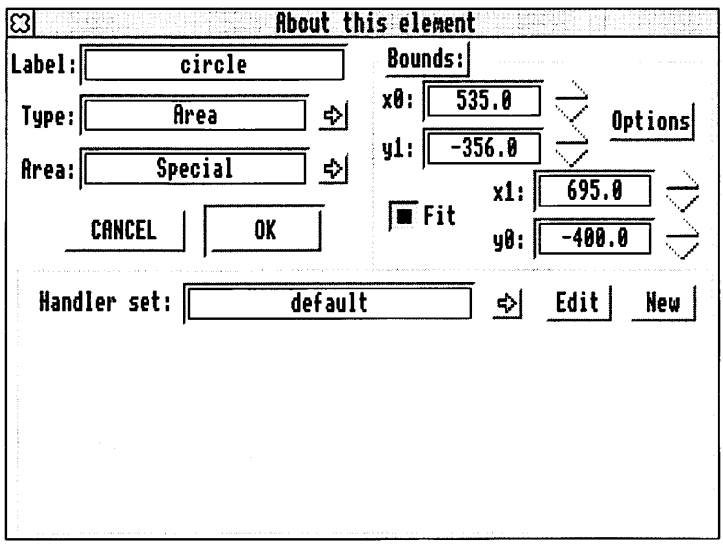
Redraw handlers

Because your application will be running in a window environment there are certain constraints applied when using graphics. Graphics must be displayed in a window otherwise they will interfere with other applications in the desktop. All objects displayed in a window must be drawn when the operating system asks you to redraw the work area of the window. S-Base does this on your behalf. Sometimes you may wish to draw your own graphics in addition to the objects in the card or table. For this reason S-

Base supports redraw handlers.

A redraw handler is called after S-Base has finished redrawing your card. At this point you can draw your own graphics on top of those drawn by S-Base.

Load the `address` format into the format layout editor. Create an area element labelled `circle`.



Select an area type of 'special'. S-Base will ignore the special area type and not display it in the card. This element will be used to position a graphic on the card.

Create a new handler set labelled `address card` and attach it to the `address` format. The redraw handler will only work if it is attached to a format because the redraw operation involves the entire format. Create a handler labelled `address redraw` attached as an S Redraw handler to the handler set and enter the following lines of code:

```
local a
dim i a[4]
card element box ahandle,"circle",a
gcol 11
circle fill (a[0] + a[2]) / 2,(a[1] + a[3]) / 2,(a[2] -
a[0]) / 2
```

This code will first create a 4 byte integer array of 4 elements. This array will be used to find the graphic coordinates of an element in the card. The array must be at least this size and use 4 byte integers.

The **card element box** command will return the coordinates in OS units of an element in a card. The command has three arguments: the first argument is the handle of the card; the second argument is the label of the element; the third argument is an array variable for S-Base to store the coordinates in.

The **gcol** command changes the graphics colour to red (11) although this depends on the screen mode and palette. The **circle fill** command will draw a circle in the centre of the element with a radius of half the width of the element.

Run your application and open the `address` card. You should see a red circle where the special area was.

27. Accessing RISC OS

Sys

S-Base provides a single command to interface with RISC OS. **sys** is the universal command to access all the facilities of the operating system. The **sys** command has a number of arguments. The first argument is the operating system call. Each routine in the operating system has a unique system call name and number with which the operating system identifies it.

The remaining arguments are registers used to store information used by the routine. The number of registers used depends on the operating system routine being called. If the routine returns results the **to** command is used to store these in another set of registers. Try entering the following lines of code into the command line:

```
s:sys "OS_WriteC",20
```

The system call `OS_WriteC` is the operating system universal output routine. The line above writes the ASCII character `20` to the vdu driver. ASCII 20 is the code to reset the computer's palette. You will need to select the default desktop palette from the palette utility on the icon bar.

The following system call (`OS_ReadC`) reads a keypress and returns the character's ASCII code in register 0.

```
s:sys "OS_ReadC" to a
s:print @chr(a)
g
```

The **sys** command is of little use to the average user. Only the experienced programmer should attempt to directly use the operating system routines. S-Base has implemented many of these routines as commands and functions which you can easily access.

When the operating system returns a string it will return it as a block of ASCII codes. This will need to be converted into a string. The **@stringize** function will return a string from an array of integers. The string must be less than 254 characters. In practice **@stringize** has little use.

A list of the RISC OS calls and their arguments are documented in the RISC OS Programmers' Reference Manual.

Error handling

S-Base provides three methods of handling errors. The method it will use depends on the *error state* of the application. The error state of an application can be one of three values.

If the error state is set to 0, S-Base will report the error and load the offending item into an editor. The execution of the application will be terminated.

If the error state is set to 1, S-Base will call an error handler. An error handler is a section of code inside a program, procedure, handler or menu dedicated to processing errors. The **def local error** and matching **enddef** commands are used to instruct S-Base which lines of the code are for error handling. This code will be ignored until an error occurs. If S-Base encounters a **def local error** command it will automatically set the error state to 1.

S-Base will immediately go to the error handler code when an error occurs. The functions **@errnum** and **@errstr** will contain the number and message of the error. The error handler should use these functions to process the error and/or notify the user.

The execution of the application will not terminate. S-Base will continue executing code past the end of the error handler as if nothing has happened. For example,

```
def proc hello
   local fhandle
   def local error
      error 1,@errstr
      return FALSE
   enddef
   fhandle = @openout "ADFS::0.$.data.hello"
   print to fhandle,"hello"
   close fhandle
   return TRUE
enddef
```

This code will write the string `"hello"` to the file `hello`. If an error occurs at any stage S-Base will go to the error handler. It will display an error message and return `FALSE`. You should be familiar with the **error** command by now so it will not be commented on. The **return** command will prevent S-Base from attempting to open the file again when it passes through the **enddef**. The procedure returns `TRUE` if it was successful in

writing to disc and FALSE if it failed.

If the error state is set to 2, S-Base will set **@errnum** and **@errstr** and continue executing the code from the next line. The error will need to be handled where the error occurred.

The **error set** command will set the error state of an application. The single argument for this command is the error state number.

28. Archiving and Importing applications

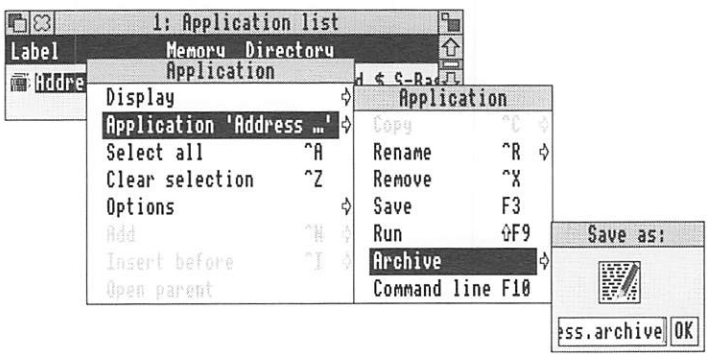
The applications you will create with S-Base are valuable assets. A considerable amount of time, energy and inspiration goes into creating an application. It is important to ensure this is not suddenly lost in an accident. S-Base provides maintenance facilities for archiving and importing applications.

Archiving

The S-Base archive function will generate a text archive of an application's resources. The archive file will contain a detailed description of every object in the application. S-Base can use this description to rebuild the application, if necessary.

The archive function does not archive the data stored in the files resource of an application. You will have to backup the 'data' directory of the application by other means.

Open the application list and select 'Archive' from the application menu.



Drag the text icon in the 'Save as:' window to a directory or text editor.

The default pathname for the archive will save it inside the application's directory. Any unsaved data in open text editors will not be saved as part of the archive.

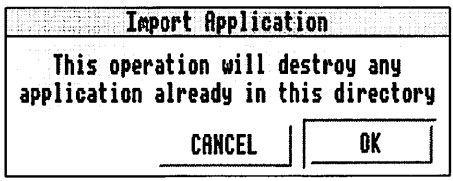
If you are upgrading to a new version of S-Base your applications will need to be archived in the old version first. The way S-Base stores applications may alter in the future, so S-Base will probably need to

rebuild them from the archive files. In this situation click on 'OK' in the save window and S-Base will save the archive inside the application.

Importing

The S-Base import facility allows an application to be restored from an archive file.

Remove your address application from the application list by selecting 'Remove' in the application menu or pressing Ctrl-X. Reload the application holding down Ctrl (use Ctrl-double-click instead of double-click).



S-Base will prompt you to confirm the import operation. Click 'OK'.

S-Base will look inside the application's directory for a text file labelled `archive`. Next it will destroy the existing resources and rebuild them from this file.

If you have been given an archive by a friend, you will need to create a blank (new) application and place the archive file inside it. Import this application to generate the resources from the archive file.

Index

! 49
$ 52
&& 49
& 50
<< 50
>>> 51
>> 51
|| 49
| 50
~ 50
@abs 154
@acos 153
@addr 173
@altered 171
@asc 158
@asin 153
@atan 153
@atanq 153
@bget 215
@bitsetcount 178
@bittest 179
@bof 213
@chr 158
@clickelement 171
@clickelementlabel 105, 171
@clickmouse 171
@cos 153
@cosh 153
@cursor 171
@date 149
@datesi 149
@datesn 149
@deg 154
@deleted 213
@dollar 154
@elaltered 171
@element 172
@elementhandle 175
@elementlabel 172

@elementvalue 124, 169, 172
@entry 142-143, 172
@entryaltered 172
@entrycount 199
@entrytext 188
@errnum 226
@errstr 226
@eventtype 173
@exp 153
@ext 199, 216, 220
@formstr 158
@handle 170, 172, 199
@indexcount 213
@instr 158
@int 154
@justaltered 124, 172
@key 172
@lastentry 143, 172
@left 158
@len 158
@ln 153
@loadindex 213
@log 153
@match 158
@max 154
@mean 154
@menu 172
@menufind 172, 184-185
@menuselect 172, 187
@mid 158
@min 154, 199
@msg 158
@mult 159
@onespace 159
@openin 214, 219
@openout 214
@openup 215
@order 213
@pad 159
@pi 155

@point 221
@previous 117
@ptr 215, 220
@rad 154
@record 130
@recordcount 106
@recsize 213
@right 159
@rnd 155
@round 155
@sdev 154
@selectedhandle 178
@sgn 155
@sin 153
@sinh 153
@spc 159
@sqr 52, 155
@str 159
@string 159
@stringize 159, 225
@sum 154
@tan 153
@tanh 154
@time 146, 149
@timedm 150
@timedom 150
@timedow 149
@timedoy 150
@timefod 150
@timefom 150
@timefoy 150
@timeh 149
@timemoy 150
@timemy 150
@times 149
@timesi 149
@timesn 149
@timest 149
@timeu 149
@timey 150

@todate 149
@tof 213
@tolower 159
@totime 149
@toupper 159
@trim 124, 159
@val 159
@vrecsize 213

A

Acorn Replay 220
Action style 68
Adjust 2
AFG 82
AFG dialogue 82, 135, 191
Align/Distribute dialogue 77
Altering icons 78
Applications 5
Archive 215
Area 12
Area tool 86
Arithmetic operators 49
Arithmetic shift right 51
Array delete 201
Arrays 46
Auto size 83
Automatic format generation 81
Automatic recalc 89

B

Bitstring 178
Bitwise AND 50
Bitwise NOT 50
Bitwise operators 49
Bitwise OR 50
Border 88
Box style 68
Box header style 68
Bput 215
Bring to front 85, 142

Browser 67
Buffer load 219
Buffer load ext 219
Buffer new 219
Buffer save 220
Buffer set ext 219

C

Cancel style 68
Card dispose 97
Card element box 223
Card exit poll 100
Card hide 179
Card new 97
Card poll 97, 197
Card poll until 214
Card set element 107
Card set protect 128
Card set table 197
Card show 97
Card update 105
Card update valuebar 124
Cards 12
Case 104, 112
CD-ROM 60
Circle 221
Clear 53
Clg 221
Click 2
Close 122, 215
Close window 22
Colour selection dialogue 39
Command line 2, 21, 25, 44
Comment 96
Configure 22
Confirmed deletion 31
Constants 14, 40
Controlling S-Base 95
Conventions 2
Copying between applications 34

Copying between windows 34
Copying resources 30
Create new style 71
Creating a new application 23
Creating files 58
Creating formats 81
Creating resources 36
Creation tool 72, 76
Ctrl-double-click 2
Current object 171

D

Data 5-6
Data type dialogue 56
Data types 7, 56
Date 8, 44
Def local error 226
Def proc 15, 111, 113
Default style 67-68
Default action style 68
Delete 212
Deleting resources 31
Developers Environment 1
Dim 46
Disc files 60
Disc heap 57, 217
Display options 86
Double-click 2
Dragging 33
Draw 221
Drawing 12, 20
Drawing tool 85

E

Editing while S-Base is running 96
Element info dialogue 87
Element options dialogue 88
Element ordering 141
Element spacing 83
Element spacing dialogue 83

Elements 11, 46
Else 106
Endcase 105, 112
Enddef 15, 111, 113, 226
Endwhile 110, 112
Error 126
Error dialogue 126
Error set 227
Error state 226
Events 13
Exclusive OR 50
Execution context 95
Exit 189
Export 220
Expression 44, 47
External 8, 57, 217
External field 217

F

Ffor 109, 112
Field 11
Field margin 83
Field spacing 83
Field tool 85
Fields 9
File field dialogue 61
File fields 55
File info dialogue 59
File type 60
Filename 60
Files 9, 55
Filters 211
Fixed length records 10
Float 8, 43
Floating baseline 89
Floating structures 60
Flush 212
Font style dialogue 38
Font styles 18, 38
For 108, 112

Format 81
Format display dialogue 139
Format element types 11
Format info dialogue 101
Format layout editor 84
Formats 11, 65
FormEd 65
Full info 27
Functions 52

G

Gcol 221
Grid 74-75, 86
Grid dialogue 75

H

Handler set dialogue 98
Handler set item dialogue 98
Handler sets 13, 16
Handlers 16
Horizontal alignment 88

I

Icon 11
Icon info dialogue 76
Icon styles 67
Icon tool 86
Iconize 29
Icons 66
If 106, 112
Import 215, 220
Index element dialogue 116
Index info dialogue 115
Indexes 6
Infinite loop 110
Info 22
Info bar 72-73, 86
Input focus 40
Insert before 36
Installation 3

Integer 8, 43
Interactive help 83

K

Keyboard short cuts 40
Keypress handlers 124

L

Label 5
Large icon 27
Left arrow style 68
Line 12
Line position dialogue 93
Line spacing 83
Line tool 85
Load 105
Load event 173
Local 45, 108
Local variable 45, 111
Logical AND 49
Logical NOT 49
Logical operators 49
Logical OR 49
Logical shift right 51
Lvar 53

M

Memory files 60
Menu 2
Menu clear protect 186
Menu clear select 186
Menu dispose 188
Menu insert 184
Menu maker handlers 185
Menu new 184
Menu select handler 187
Menu set 185, 188
Menu set menu 185
Menu set protect 186
Menu set select 186

Menus 17
Move 34, 221
Move window 74
Multi-column reports 163
Multi-entry reports 166

N

Nested cards 122
Network directory 23
New application 23
New application dialogue 23
Next 109, 112

O

Ok style 68
Open window 22
Operators 47
Option style 69
Origin 221
OS Units 75
Otherwise 105

P

Pack 212
Paper options 86
Paper options dialogue 138
Parent 29
Physical order 28, 34
Plot 221
Plunder 73
Pmenu dispose 134
Pmenu hide 134
Pmenu insert 134
Pmenu new 134
Pmenu poll 134
Pmenu show 134
Point 221
Popup style 69
Positioning icons 77
Print to 215

Printing viewers 36
Priority 47
Problems dialogue 32
Proc 111
Procedure 111
Procedures 14
Program lists 16
Program templates 103
Programs 15
Prompt style 11, 69
Prompt margin 83
Prompt tool 85
Protected 88
Protected fields 130
Put to back 85, 142

Q

Quit 23

R

Radio style 69
Read only 10, 60
Record size 9, 62
Records 9
Rectangle 12, 221
Rectangle tool 85
Reindex 213
Remove an application 24
Remove contexts 22
Rename dialogue 30
Renaming resources 29
Renumber 73
Reordering resources 33
Repeat 109, 112
Report 135
Report dispose 143
Report entry 137
Report exit run 143
Report new 143
Report print 143

Report run 143
Report set entry 167
Report show 143
Report title 143
Reports 13
Request entry handler 165
Request entry handlers 142
Resize window 74
Resizing icons 79
Resizing templates 80
Resources 5
Return 111
Revert to style 71
Right arrow style 69
RISC OS printer driver 138
Run 95

S

S 1
S Click 99, 169
S Enter entry 169
S Enter field 170
S Exit entry 169
S Exit field 170
S Extension 170
S Keypress 169
S Menu maker 170
S Menu select 171
S Redraw 171
S Request entry 169
S-Base 1
S-Base handlers 169
Save 212
Scared cards 121
Scroll window 74
Select 2, 210
Selecting objects 29
Selection tool 72, 84
Selector 12
Selector insert 175

Selector tool 85
Set ptr 215, 219
Shift left 50
Shift-double-click 2
Side effects 45, 133
Small icon 27
Sprite action style 69
Sprite tool 85
Sprites 11, 19
Step 108
String 8, 43
Structure 8, 44, 57
Style 66
Style list 73
Style overrides 78
Submenus 18
Subscript 46
Sys 225
SysMerge dialogue 3
System field 62

T

Table clear protect 197
Table delete 200
Table ext 198
Table insert 200
Table new 170
Table set ext 198
Table set protect 197
Tables 13, 45
Template display options 73
Template info dialogue 70
Template layout editor 71
Templates 13, 65
Text editors 38
Text only 13, 75
Then 106, 112
Time 8, 44
Title bar 66
Tool bar 72

Try out template 74
Type mismatch error 48

U

Until 109, 112
Update 107
Update style 71
Updating !System 3
Use 207

V

Value bar 82
Variable length 60
Variable length records 10
Variables 43
Vdu 221
Viewers 27

W

When 104, 112
While 110, 112
Window numbering 28
Window styles 67
Windows 65
Writeable style 69

Z

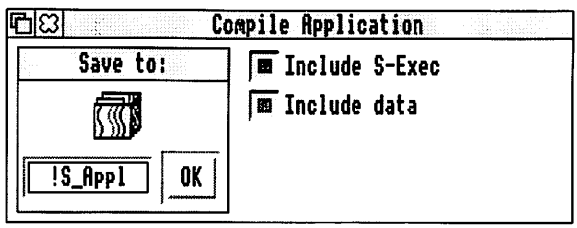
Zap 213
Zero 97
Zero length 60

Compiling Applications

S-Base Developer allows you to compile S-Base applications into stand-alone RISC OS applications. You may then distribute this application without any royalty payment for the run-time system, S-Exec.

How to compile

When your application is complete, select 'Compile' from the application list viewer menu.



There are two options available. Firstly, you can choose to include the run-time system (S-Exec) with the application. If S-Exec is included, the application will use the copy of S-Exec inside it.

If this option is off, S-Exec will not be included in the application. This reduces the size of the application by 170k, but the application will not run unless S-Exec (i.e. an application containing it) has be seen by the filer. Normally you would compile one application with S-Exec, and compile all the other without it, to save disc space.

The second option copies the data directory into the compiled application. This means that the compiled application will start with the same data as your development application. This is useful if you have default data which the compiled application should start with.

Select the options required then type in a filename for the compiled application. This need not be the same as the development application although it usually is.

Drag the application icon to the destination directory. S-Base will copy in S-Exec and the data if required, then proceed to compile the application.

The application's resources are stored in an intermediate form in the 'execres' file within the application directory. This file is unreadable and

cannot be loaded into S-Base. Therefore you can send a compiled application knowing that your work will not be copied or modified.

Fidelity

S-Exec 1.0 can execute all applications generated by S-Base 1.0. There are no known incompatibilities. However, there are differences of operation which you should be aware of.

Error handling

S-Exec does not handle errors in the same way as S-Base since it does not have access to the development application. Resources such as handlers and programs are in intermediate code and there is no way for S-Exec to tell you where an error occurred. Also, to save space S-Exec does not include error messages; it merely provides an error number. To locate the error, you should repeat the problem within S-Base using the development application.

For this reason, if you intend to distribute a compiled application you should keep a backup of the development application used to compile it. Reported problems with the compiled application can then be located and fixed easily.

Your application should provide an error handler, so that when an error occurs the application doesn't just report it and exit. The error handler should report the error as '*An unexpected error has occurred in <application name> (type <type>)*', where <type> is **@errnum**. If possible it should close any open cards/tables and continue execution. If further execution is not possible, inform the user '*<application name> has suffered a fatal internal error (type <type>) and must exit immediately*' and use the **exit** command. S-Exec will ensure that all files are closed correctly in this situation.

Data

If when running an application, S-Base detects that a data or index file has altered, it will be imported or rebuilt automatically. S-Exec cannot do this, and assumes that all data and index files are correct if present.

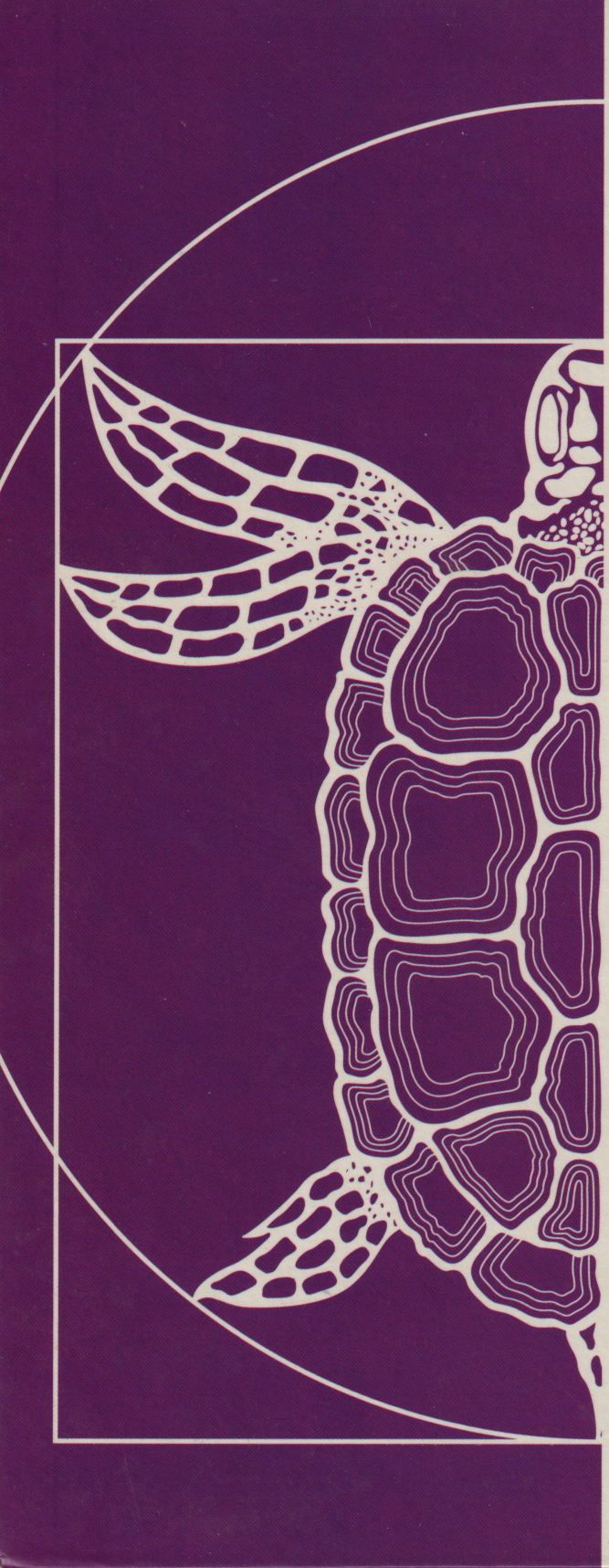
You should ensure that the compiled application either has no data files, or

has up-to-date data files. The same applies to indexes. S-Exec assumes that any data or index files which are present are usable without any processing. Unexpected results will occur if you provide a compiled application with old data or index files.

Like S-Base, S-Exec can rebuild indexes and create data files if they are not present.

Configuration

S-Exec uses the same configuration file as S-Base ('config') but as you would expect, options which apply only to the SDE have no effect on S-Exec.



Longman Logotron

124 Cambridge Science Park
Milton Road
Cambridge CB4 4ZS

Tel. (0223) 425558
Fax (0223) 425349